

USENIX

conference

proceedings

Proceedings of the 16th USENIX Security Symposium

# 16th USENIX Security Symposium

Boston, MA, USA  
August 6–10, 2007

Sponsored by  
The USENIX Association

**USENIX**  
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Boston, MA, USA, August 6–10, 2007

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA  
Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • <http://www.usenix.org>

The price is \$45 for members and \$55 for nonmembers.  
Outside the U.S.A. and Canada, please add  
\$20 per copy for postage (via air printed matter).

### Thanks to Our Sponsors

Google™

Microsoft®  
**Research**

 symantec

### Thanks to Our Media Sponsors

*ACM Queue*  
*Addison-Wesley Professional/*  
*Prentice Hall Professional*  
*Dr. Dobb's Journal*  
*Free Software Magazine*  
*GRIDtoday*

*HPCwire*  
*IEEE Security & Privacy*  
*ITtoolbox*  
*Linux Journal*  
*Linux Pro Magazine*  
*No Starch Press*

*Penton TechX*  
*SNIA*  
*StorageNetworking.org*  
*UserFriendly.org*

© 2007 by The USENIX Association  
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-931971-54-4

**USENIX Association**

**Proceedings of the  
16th USENIX Security Symposium**

**August 6–10, 2007  
Boston, MA, USA**

## Conference Organizers

### Program Chair

Niels Provos, *Google Inc.*

### Program Committee

Kostas Anagnostakis, *Institute for Infocomm Research, Singapore*

Dan Boneh, *Stanford University*

Hao Chen, *University of California, Davis*

Monica Chew, *Google Inc.*

David Dagon, *Georgia Institute of Technology*

Marius Eriksen, *Google Inc.*

Kevin Fu, *University of Massachusetts Amherst*

Tal Garfinkel, *Stanford University*

Thorsten Holz, *University of Mannheim*

Somesh Jha, *University of Wisconsin*

Tadayoshi Kohno, *University of Washington*

Christopher Kruegel, *Technical University Vienna*

Wenke Lee, *Georgia Institute of Technology*

Patrick McDaniel, *Pennsylvania State University*

Fabian Monrose, *Johns Hopkins University*

Vern Paxson, *ICS/LBNL*

Adrian Perrig, *Carnegie Mellon University*

Vassilis Prevelakis, *Drexel University*

Dug Song, *Arbor Networks*

Angelos Stavrou, *Columbia University*

Rebecca Wright, *Stevens Institute of Technology*

Paul Van Oorschot, *Carleton University*

Wietse Venema, *IBM Research*

Yi-Min Wang, *Microsoft Research, Redmond*

### Invited Talks Committee

Bill Aiello, *University of British Columbia*

Angelos Keromytis, *Columbia University*

Gary McGraw, *Cigital*

### The USENIX Association Staff

## External Reviewers

Salman Abdul

Eytan Adar

Elli Androulaki

Vijay Balasubramanian

Lujo Bauer

Ulrich Bayer

Michael Becher

Zina Benenson

Emery Berger

Andrew Bortz

Tanya Bragin

Timo Burkard

Pei Cao

Martin Casado

Sambuddho Chakravarty

Ranveer Chandra

Shuo Chen

Jim Chow

Mihai Christodorescu

Jim Cipar

Gabriela Cretu

Weidong Cui

Benessa Defend

William Enck

Jason Franklin

Felix Freiling

Jonathan Giffin

Ramakrishna Gummadi

Boniface Hicks

Dan Holcomb

Jason Holt

Francis Hsu

Sotiris Ioannidis

Collin Jackson

Xuxian Jiang

Rob Johnson

Chris Karlof

Sam King

Scott Knight

Maxwell Krown

Louis Kruger

Chao Liu

Xiaomin Liu

Ben Livshits

Michael Locasto

Ken MacInnis

Mohammad Mannan

Lorenzo Martignoni

Damon McCoy

Jonathan McCune

Alain Meyer

Martin Mink

Nagendra Modadugu

Andreas Moser

Ira Moskowitz

Onur Mutlu

Deholo Nali

Jose Nazario

Jon Oberheide

Adam O'Donnell

Bryan Parno

Moheeb Rajab

Shai Rubin

Reiner Sailer

Arvind Seshadri

Umesh Shankar

Weidong Shao

Alex Sherman

Randy Smith

Paul Snyder

Anil Somayaji

Yingbo Song

Pratap Subrahmanyam

Parisa Tabriz

Patrick Traynor

David Turner

Chad Verbowski

Dave Whyte

Gilbert Wondracek

Benson Wu

Glenn Wurster

Yinglian Xie

Dingbang Xu

Fang Yu

**16th USENIX Security Symposium**  
**August 6–10, 2007**  
**Boston, MA, USA**

<b>Index of Authors</b> .....	v
<b>Message from the Program Chair</b> .....	vii

**Wednesday, August 8**

**WWW Security**

SIF: Enforcing Confidentiality and Integrity in Web Applications .....	1
<i>Stephen Chong, K. Vikram, and Andrew C. Myers, Cornell University</i>	
Combating Click Fraud via Premium Clicks .....	17
<i>Ari Juels, RSA Laboratories; Sid Stamm, Indiana University, Bloomington; Markus Jakobsson, Indiana University, Bloomington, and RavenWhite Inc.</i>	
SpyProxy: Execution-based Detection of Malicious Web Content .....	27
<i>Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble, and Henry M. Levy, University of Washington</i>	

**Privacy**

Language Identification of Encrypted VoIP Traffic: Alejandra y Roberto or Alice and Bob? .....	43
<i>Charles V. Wright, Lucas Ballard, Fabian Monrose, and Gerald M. Masson, Johns Hopkins University</i>	
Devices That Tell on You: Privacy Trends in Consumer Ubiquitous Computing .....	55
<i>T. Scott Saponas, Jonathan Lester, Carl Hartung, Sameer Agarwal, and Tadayoshi Kohno, University of Washington</i>	
Web-Based Inference Detection .....	71
<i>Jessica Staddon and Philippe Golle, Palo Alto Research Center; Bryce Zimny, University of Waterloo</i>	

**Authentication**

Keep Your Enemies Close: Distance Bounding Against Smartcard Relay Attacks .....	87
<i>Saar Drimer and Steven J. Murdoch, Computer Laboratory, University of Cambridge</i>	
Human-Seeded Attacks and Exploiting Hot-Spots in Graphical Passwords .....	103
<i>Julie Thorpe and P.C. van Oorschot, Carleton University</i>	
Halting Password Puzzles: Hard-to-break Encryption from Human-memorable Keys .....	119
<i>Xavier Boyen, Voltage Security</i>	

## Thursday, August 9

### Threats

- Spamscatter: Characterizing Internet Scam Hosting Infrastructure ..... 135  
*David S. Anderson, Chris Fleizach, Stefan Savage, and Geoffrey M. Voelker, University of California, San Diego*
- Exploiting Network Structure for Proactive Spam Mitigation ..... 149  
*Shobha Venkataraman, Carnegie Mellon University; Subhabrata Sen, Oliver Spatscheck, and Patrick Haffner, AT&T Research; Dawn Song, Carnegie Mellon University*
- BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation ..... 167  
*Guofei Gu, Georgia Institute of Technology; Phillip Porras, Vinod Yegneswaran, and Martin Fong, SRI International; Wenke Lee, Georgia Institute of Technology*

### Analysis

- Integrity Checking in Cryptographic File Systems with Constant Trusted Storage ..... 183  
*Alina Oprea and Michael K. Reiter, Carnegie Mellon University*
- Discoverer: Automatic Protocol Reverse Engineering from Network Traces ..... 199  
*Weidong Cui, Microsoft Research; Jayanthkumar Kannan, University of California, Berkeley; Helen J. Wang, Microsoft Research*
- Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation ..... 213  
*David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song, Carnegie Mellon University*

### Low Level

- OSLO: Improving the Security of Trusted Computing ..... 229  
*Bernhard Kauer, Technische Universität Dresden*
- Secretly Monopolizing the CPU Without Superuser Privileges ..... 239  
*Dan Tsafir, The Hebrew University of Jerusalem and IBM T.J. Watson Research Center; Yoav Etsion and Dror G. Feitelson, The Hebrew University of Jerusalem*
- Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems ..... 257  
*Thomas Moscibroda and Onur Mutlu, Microsoft Research*

## Friday, August 10

### Obfuscation

- Binary Obfuscation Using Signals ..... 275  
*Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews, The University of Arizona*
- Active Hardware Metering for Intellectual Property Protection and Security ..... 291  
*Yousra M. Alkabani and Farinaz Koushanfar, Rice University*

### Network Security

- On Attack Causality in Internet-Connected Cellular Networks ..... 307  
*Patrick Traynor, Patrick McDaniel, and Thomas La Porta, The Pennsylvania State University*
- Proximity Breeds Danger: Emerging Threats in Metro-area Wireless Networks ..... 323  
*P. Akritidis, Computer Laboratory, Cambridge University; W.Y. Chin, Institute for Infocomm Research (I<sup>2</sup>R), Singapore; V.T. Lam, University of California, San Diego; S. Sidiroglou, Columbia University; K.G. Anagnostakis, Institute for Infocomm Research (I<sup>2</sup>R), Singapore*
- On Web Browsing Privacy in Anonymized NetFlows ..... 339  
*S.E. Coull, Johns Hopkins University; M.P. Collins, Carnegie Mellon University; C.V. Wright and F. Monroe, Johns Hopkins University; M.K. Reiter, Carnegie Mellon University*

## Index of Authors

Agarwal, Sameer . . . . .	55	Gribble, Steven D. . . . .	27	Newsome, James . . . . .	213
Akritidis, P. . . . .	323	Gu, Guofei . . . . .	167	Oprea, Alina . . . . .	183
Alkabani, Yousra M. . . . .	291	Haffner, Patrick. . . . .	149	Popov, Igor V. . . . .	275
Anagnostakis, K.G. . . . .	323	Hartung, Carl . . . . .	55	Porras, Phillip . . . . .	167
Anderson, David S. . . . .	135	Jakobsson, Markus . . . . .	17	Reiter, Michael K. . . . .	183, 339
Andrews, Gregory R. . . . .	275	Juels, Ari. . . . .	17	Saponas, T. Scott . . . . .	55
Ballard, Lucas. . . . .	43	Kannan, Jayanthkumar . . . . .	199	Savage, Stefan . . . . .	135
Boyen, Xavier. . . . .	119	Kauer, Bernhard . . . . .	229	Sen, Subhabrata . . . . .	149
Bragin, Tanya . . . . .	27	Kohno, Tadayoshi. . . . .	55	Sidirolou, S. . . . .	323
Brumley, David. . . . .	213	Koushanfar, Farinaz . . . . .	291	Song, Dawn . . . . .	149, 213
Caballero, Juan . . . . .	213	La Porta, Thomas . . . . .	307	Spatscheck, Oliver . . . . .	149
Chin, W.Y. . . . .	323	Lam, V.T. . . . .	323	Staddon, Jessica . . . . .	71
Chong, Stephen . . . . .	1	Lee, Wenke . . . . .	167	Stamm, Sid . . . . .	17
Collins, M.P. . . . .	339	Lester, Jonathan . . . . .	55	Thorpe, Julie. . . . .	103
Coull, S.E. . . . .	339	Levy, Henry M. . . . .	27	Traynor, Patrick . . . . .	307
Cui, Weidong . . . . .	199	Liang, Zhenkai . . . . .	213	Tsafir, Dan. . . . .	239
Debray, Saumya K. . . . .	275	Masson, Gerald M. . . . .	43	van Oorschot, P.C. . . . .	103
Déville, Damien . . . . .	27	McDaniel, Patrick. . . . .	307	Venkataraman, Shobha. . . . .	149
Drimer, Saar . . . . .	87	Monrose, Fabian. . . . .	43, 339	Vikram, K. . . . .	1
Etsion, Yoav . . . . .	239	Moscibroda, Thomas . . . . .	257	Voelker, Geoffrey M. . . . .	135
Feitelson, Dror G. . . . .	239	Moshchuk, Alexander. . . . .	27	Wang, Helen J. . . . .	199
Fleizach, Chris . . . . .	135	Murdoch, Steven J. . . . .	87	Wright, Charles V. . . . .	43, 339
Fong, Martin. . . . .	167	Mutlu, Onur . . . . .	257	Yegneswaran, Vinod. . . . .	167
Golle, Philippe . . . . .	71	Myers, Andrew C. . . . .	1	Zimny, Bryce . . . . .	71



## Message from the Program Chair

Welcome to the 16th USENIX Security Symposium!

In these proceedings, you will find the 23 papers presented at the symposium, held on August 8–10 in Boston, MA. These papers were selected from a total of 187 paper submissions covering a broad range of computer and network security. The number of submissions is a new record for USENIX Security, and it led to an acceptance rate of only 12%. Although this makes USENIX Security very selective, it also means that we had to reject many very good submissions which will most likely appear in other venues.

A program committee of 26 experts in the field reviewed and discussed the papers in a two-day meeting in Mountain View. Each paper received at least 3 reviews, and the program committee members wrote over 600 reviews. The best papers were selected based on their scientific contribution and technical quality. The identities of the reviewers was not revealed to the authors. For some papers, we also sought advice from experts outside the program committee. The names of all external reviewers are printed at the beginning of these proceedings.

In addition to the papers printed here, the symposium program included tutorials, invited talks, a keynote speech by Steven Levy, a panel on cellular network security, and a work-in-progress session.

I would like to express my gratitude to the program committee members for their hard work: on average, each PC member reviewed 23 papers, generating more than 140 lines of review per submission. Of course, this would not have been possible without the efforts of the authors submitting their papers. Especially for the papers we were not able to accept, I hope that the authors found the comments helpful. I would like to point out that among the papers we received were many in which the authors invested a lot of time building and evaluating systems. Although it has become increasingly challenging to conduct research and publish on systems, I hope that USENIX Security will continue its tradition of being one of the top systems venues where applied research that has impact is presented.

It goes without saying that the USENIX staff were instrumental in making this year's USENIX Security Symposium a success. I am especially grateful for their help in every aspect of organizing the event, including the call for papers, the preparation of the proceedings, the logistics of the PC meeting, and all the rest that make a conference a success. I would also like to thank Bill Aiello, Angelos Keromytis, and Gary McGraw for organizing the invited talks track, Martin Casado for organizing the work-in-progress session, and Radu Sion for organizing the poster session.

**Niels Provos, Google Inc.**  
Security '07 Program Chair



# SIF: Enforcing Confidentiality and Integrity in Web Applications

Stephen Chong      K. Vikram      Andrew C. Myers  
*Department of Computer Science, Cornell University*

## Abstract

SIF (Servlet Information Flow) is a novel software framework for building high-assurance web applications, using language-based information-flow control to enforce security. Explicit, end-to-end confidentiality and integrity policies can be given either as compile-time program annotations, or as run-time user requirements. Compile-time and run-time checking efficiently enforce these policies. Information flow analysis is known to be useful against SQL injection and cross-site scripting, but SIF prevents inappropriate use of information more generally: the flow of confidential information to clients is controlled, as is the flow of low-integrity information from clients. Expressive policies allow users and application providers to protect information from one another.

SIF moves trust out of the web application, and into the framework and compiler. This provides application deployers with stronger security assurance.

Language-based information flow promises cheap, strong information security. But until now, it could not effectively enforce information security in highly dynamic applications. To build SIF, we developed new language features that make it possible to write realistic web applications. Increased assurance is obtained with modest enforcement overhead.

## 1 Introduction

Web applications are now used for a wide range of important activities: email, social networking, on-line shopping and auctions, financial management, and many more. They provide services to millions of users and store information about and for them. However, a web application may contain design or implementation vulnerabilities that compromise the confidentiality, integrity, or availability of information manipulated by the application, with financial, legal, or ethical implications. According to a recent report [33], web applications account for 69% of Internet vulnerabilities. Current techniques appear inadequate to prevent vulnerabilities in web applications.

In general, information security vulnerabilities arise from inappropriate information dependencies, so tracking information flows within applications offers a comprehensive solution. Confidentiality can be enforced

by controlling information flow from sensitive data to clients; integrity can be enforced by controlling information flow from clients to trusted information—as a side effect, protecting against common vulnerabilities like SQL injection and cross-site scripting. In fact, recent work [14, 19, 37, 15] on static analysis of PHP and Java web applications has used dependency analyses to find many vulnerabilities in existing web applications and web application libraries. Dynamic tainting can detect some improper dependencies and has also proved useful in detecting vulnerabilities [39, 6]. However, static analyses have the advantage that they can conservatively identify information flows, providing stronger security assurance [28].

Therefore, we have developed Servlet Information Flow (SIF), a novel framework for building web applications that respect explicit confidentiality and integrity information security policies. SIF web applications are written in Jif 3.0, an extended version of the Jif programming language [21, 24] (which itself extends Java with information-flow control). The enforcement mechanisms of SIF and Jif 3.0 track the flow of information within a web application, and information sent to and returned from the client. SIF reduces the trust that must be placed in web applications, in exchange for trust in the servlet framework and the Jif 3.0 compiler—a good bargain because the framework and compiler are shared by all SIF applications.

The security policies used in SIF are both strong and expressive. Information flow is tracked through a type system that tracks all information flows, not merely explicit flows. Security enforcement is *end-to-end*, because policies are enforced on information from when it enters the web application, to when it leaves, even as information flows between different client requests. The security policies are expressive, allowing complex security requirements of multi-user systems to be enforced. Unlike prior frameworks for tracking information flow in web applications, policies can express fine-grained requirements for *both* confidentiality and integrity. Further, the interactions between confidentiality and integrity are controlled.

The end-to-end security provided by information-flow control has long been appealing, but much theoretical work on language-based information flow has not yet

been successfully put into practice. We have identified limitations of existing security-typed languages for reasoning about security in a dynamic external environment, and we have extended the Jif language with new features supporting these dynamic environments, resulting in a new version of the language, Jif 3.0.

Information-flow control mechanisms work by labeling information. In previous information flow mechanisms, the space of labels is essentially static. In earlier versions of Jif, for example, labels are expressed in terms of principals, but the set of principals is fixed at compile time. This is a serious limitation for web applications, which often add new users at run time. Jif 3.0 adds the ability for applications to create their own principals, dynamically extending the space of information labels. Moreover, Jif 3.0 allows applications to implement their own authentication and authorization mechanisms for these application-specific principals—a necessity given the diversity of authentication schemes needed by different applications. Jif 3.0 also improves Jif’s ability to reason about dynamic security policies, allowing, for example, web application users to specify their own security requirements at run time and have them enforced by the information flow mechanisms. These new mechanisms create new information channels, but Jif 3.0 tracks these channels and prevents their misuse.

To explore the performance and usability of SIF, we developed two web applications with non-trivial security requirements: an email application specialized for cross-domain communication, and a multiuser shared calendar. Both applications add new principals and policies at run time, and both allow users to define their own information security policies, which are enforced by the same mechanisms used for compile-time policies.

In summary, this paper makes three significant contributions:

- It shows how to use language-based information flow to construct a practical framework for high-assurance web applications, in which information flow is tracked to and from clients, and users can specify and reason about information security. To our knowledge, this is the first implemented web application framework to strongly enforce both confidentiality and integrity.
- It shows that application-defined mechanisms for access control and authentication, and a dynamically extensible space of labels, can be integrated securely with language-based information flow.
- It describes the experience using these new mechanisms to build realistic web applications.

The remainder of the paper is structured as follows. Section 2 gives an overview of the Servlet Information Flow framework, including some background on Jif.

Section 3 introduces the new dynamic features in Jif 3.0, which enhance Jif’s ability to express and enforce dynamic security requirements. Our experience with building web applications in SIF is described in Section 4. Section 5 covers related work, and Section 6 concludes.

## 2 Servlet Information Flow framework

SIF is built using the Java Servlet framework [7], but presents a higher-level interface to web applications. Through a combination of static and dynamic mechanisms, SIF ensures that web applications use data only in accordance with specified security policies, by tracking the flow of information in the server, and information sent to and from the client. Web applications in SIF are written entirely in Jif 3.0, an extended version of the *security-typed language* [36] Jif, in which types are annotated with information flow policies. Security policies are enforced on information as it flows through the system, giving stronger security assurance than ordinary (discretionary) access control.

In designing SIF, we faced two main challenges. The first was identifying information flows in web applications, including information that flows over multiple requests. For example, a request sent to a server by a user may contain information about the user’s previous request and response. The second challenge was to restrict insecure information flows while providing sufficient flexibility to implement full-fledged web applications. The resulting framework is a principled approach to designing realistic, secure web applications.

SIF is implemented in about 4040 non-comment, non-blank lines of Java code. An additional 960 lines of Jif code provide signatures for the Java classes that web applications interact with. Jif signatures provide security annotations for Java classes, and expose only a subset of the actual methods and fields to clients. SIF web applications are compiled against the Jif signatures, but linked at run time against the Java classes. Some Java Servlet framework functionality makes reasoning about information security infeasible. Using signatures and wrapper classes, SIF necessarily limits access to this functionality, but without preventing implementation of full-fledged web applications.

In this section, we first describe the threat model that SIF addresses, and the security assurances that SIF provides. We present some background about Jif before describing the design of SIF.

### 2.1 Threat model and security assurance

**Threat model.** We assume that web application clients are potentially malicious, and that web application implementations are benign but possibly buggy. Thus, we aim to ensure that appropriate confidentiality and in-

egrity security policies are enforced on server-side information regardless of the actions of clients, or the mistakes of well-meaning application programmers.

Although the Jif programming language prevents the unintentional violation of information security, it provides mechanisms for explicit intentional downgrading of security policies (see Section 4.3). While a well-meaning programmer will be unable to accidentally misuse these mechanisms, a malicious programmer may be able to subvert them, or use certain covert channels that Jif does not track (see Section 2.2).

We do not address network threats, such as denial of service attacks, or the interception and alteration of data sent over the network.

The Jif compiler and SIF are added to the trusted computing base, which already includes the servlet container, and the software stack required to run the servlet container. Note that SIF web applications are not part of the trusted computing base, whereas in standard servlet frameworks, web applications must be trusted.

**Security assurance.** In a typical web application, security assurance consists of convincing each party with a stake in the system that the application enforces their security requirements. Obviously users would like to have assurance that information they input will be confidential, and information they view is not corrupted. The application provider (i.e., deployer) may also have confidentiality and integrity requirements for its information. Like other recent work on improving security of web applications (e.g., [14, 18, 37, 15]), we focus on providing assurance to deployers. The difference here is that SIF enforces rich policies for information integrity and confidentiality, including policies provided by the user.

Although we focus on providing assurance to deployers, it is worth considering security assurance from a web application user's perspective. Users must be convinced that they are communicating with an application that enforces their security requirements. The security validation offered by SIF effectively partitions the security assurance problem into two parts: first, ensuring that the application respects users' security requirements, and second, ensuring the server users communicate with is correctly running the application.

SIF addresses the first part of the assurance problem: verifying the security properties of web application code. SIF does not address the second part: convincing a remote client they are communicating with verified code. This step is important if the web application provider might be malicious. However, remote attestation methods [34, 10, 30] seem likely to be effective in solving this second problem. Attestation methods could be used to sign application code, or alternatively, to sign a verification certificate from a trusted SIF compiler that has checked the code. We leave integration of attestation

mechanisms till future work.

In any case, concern about malicious application providers should not be exaggerated; users' willingness to spend money via web applications suggests they already place a modicum of trust in them. This work aims to ensure this trust is justified. At a minimum, this means application deployers can be more confident in making possibly legally binding representations to their users.

The SIF framework provides the following security assurances to deployers of web applications.

- SIF applications enforce explicit information security policies. In particular, SIF ensures that information sent to the client is permitted to be read by the client, thus ensuring that confidential information held on the server is not inadvertently released to the client. Further, information received from the client is marked as tainted by the client, helping prevent inappropriate use of low-integrity information. Thus, useful confidentiality and integrity restrictions are enforced in SIF applications.
- The information security policies of back-end systems (e.g., a database, file system, or legacy application) are also enforced, provided these systems have appropriate interfaces annotated with Jif 3.0 security policies. Thus, adding a web front-end to an existing system does not weaken the security assurance of that system, modulo the assumptions of our threat model.
- Jif ensures that security policies on information are not unintentionally weakened, or *downgraded*. However, many web applications that handle sensitive information intentionally downgrade information as part of their functionality. As discussed further in Section 4.3, SIF web applications must satisfy rules that enforce *selective downgrading* [22, 26] and *robustness against all attackers* [5], security conditions that provide strong information flow guarantees in the presence of downgrading.
- SIF web applications can produce only well-formed HTML. While cascading style sheets and JavaScript may be used, they cannot be dynamically generated, and must be explicitly specified in the deployment descriptor, where they can be more easily reviewed by the application deployer. The deployer thereby gains assurance that a web application does not contain malicious client-side code.

## 2.2 Background on Jif

SIF web applications are written in Jif 3.0, a new version of the Jif programming language. To understand the design of SIF, some background on the Jif programming language is helpful. Readers familiar with Jif may skip this subsection. Details of some of the new features of

Jif 3.0 are given in Section 3.

Jif is a *security-typed language* [36]: a type has a security label  $L$  that describes restrictions on information at that type, which the compiler enforces. Security-type systems like that in Jif can enforce *noninterference*, ensuring that information labeled  $L$  can depend only on information labeled  $L$  or with a less restrictive label [28]. In other words, information cannot leak from higher to lower levels, nor can untrusted information affect trusted information. Proofs for noninterference exist for numerous security-typed languages, but not for any language as expressive as Jif. Jif labels are based on policies from the *decentralized label model* (DLM) [22], in which principals express ownership of information-flow policies.

A *principal* is an entity with security concerns, and the power to observe and change certain aspects of the system. In a web application, principals may be users of the application, user groups, or even the web application itself; SIF applications may choose which entities to model as principals. Web application principals may have different security concerns, and do not necessarily trust each other. By allowing principals to have different security policies, the DLM can express security concerns of mutually distrusting principals.

A principal  $p$  may delegate to another principal  $q$ , in which case  $q$  is said to *act for*  $p$ . The *acts-for* relation is reflexive and transitive, and is similar to the *speaks-for* relation [16]. The *acts-for* relation is needed to express trust relationships between principals, and can encode groups and roles. Jif supports a *top principal*  $\top$  able to act for all principals, and a *bottom principal*  $\perp$  that allows all principals to act for it. A principal may also grant its *authority* to code, meaning the code is trusted to perform actions such as declassification that could violate the principal's information security.

Jif labels are constructed from *reader policies* and *writer policies* [5]. A reader policy  $o \rightarrow r_1, \dots, r_n$  means that principal  $o$  owns the policy, and  $o$  permits any principal that can act for any  $r_i$  (or  $o$  itself) to read the data. For example, the reader policy  $\top \rightarrow p$  says that the top principal permits  $p$  to observe information. A writer policy  $o \leftarrow w_1, \dots, w_n$  is owned by principal  $o$ , and  $o$  has permitted any principal that can act for any of  $w_1, \dots, w_n$ , or  $o$  to have influenced ("written") the data.

Reader policies restrict to which principals information may flow, whereas writer policies describe from which principals information may have flowed. Reader policies thus describe confidentiality, and writer policies describe integrity (provenance) of information.

A Jif label is a pair of a *confidentiality policy* and an *integrity policy*, written  $\{c ; d\}$  for confidentiality policy  $c$  and integrity policy  $d$ . The set of *confidentiality policies* is formed by closing reader policies under conjunction and disjunction, denoted  $\sqcap$  and  $\sqcup$  respectively. The con-

junction of two confidentiality policies,  $c_1 \sqcap c_2$ , enforces the restrictions of both  $c_1$  and  $c_2$ . Thus, the readers permitted by  $c_1 \sqcap c_2$  is the intersection of readers permitted by  $c_1$  and  $c_2$ . Similarly, the readers permitted by the disjunction  $c_1 \sqcup c_2$  is the union of readers permitted by  $c_1$  and  $c_2$ . *Integrity policies* are formed by closing writer policies under conjunction and disjunction. Dually to confidentiality, conjunction and disjunction are respectively denoted  $\sqcap$  and  $\sqcup$ .

For example, in the label  $\{Alice \rightarrow Bob \sqcup Chuck \rightarrow Bob, Dave ; Alice \leftarrow \top\}$ , the confidentiality policy is the join of two reader policies,  $Alice \rightarrow Bob$  and  $Chuck \rightarrow Bob, Dave$ . Thus, information with this label can be read only by principals that can act for at least one of  $Alice$  or  $Bob$ , and at least one of  $Chuck$ ,  $Bob$ , or  $Dave$ ; clearly,  $Bob$  is one such principal. The integrity policy of the label consists of a single writer policy, owned by  $Alice$ , stating that  $Alice$  believes the data has been influenced only by principals able to act for  $Alice$  or the top principal  $\top$ . SIF uses confidentiality policies to restrict what information is sent to the client, and integrity policies to restrict how information received from the client is used.

Secure information flow requires that the label on a piece of information can only become more restrictive as the information flows through the system. Given labels  $L$  and  $L'$ , we write  $L \sqsubseteq L'$  if the label  $L'$  restricts the use of information at least as much as  $L$  does. To handle computations that combine information from different sources, the label  $L_1 \sqcup L_2$  imposes the restrictions of both  $L_1$  and  $L_2$ .

The types of variables and expressions in Jif programs include labels. For example, a value with type  $\text{int}\{o \rightarrow r ; \perp \leftarrow \perp\}$  is an integer with label  $\{o \rightarrow r ; \perp \leftarrow \perp\}$ : it can be read only by principals that can act for  $r$  or  $o$ , and has the lowest possible integrity. A Jif programmer may annotate the type declarations of fields, variables, and methods with labels; use of fields, variables, and methods must comply with the label annotations. For types left unannotated, the Jif compiler either chooses default labels, or automatically infers labels, thus reducing the annotation burden on the programmer.

Although a Jif programmer may annotate a program with arbitrary labels, he does not have complete control over security. Labels must be internally consistent for the program to type-check, and moreover, the labels must be consistent with security policies from the external environment. In SIF, a web application interacts with the external environment through the SIF interfaces, as well as interfaces for back-end services (e.g., databases).

Jif's type system prevents labeled information from being unintentionally *downgraded*, or assigned a less-restrictive label. Downgrading confidentiality increases the set of principals permitted to read the information,

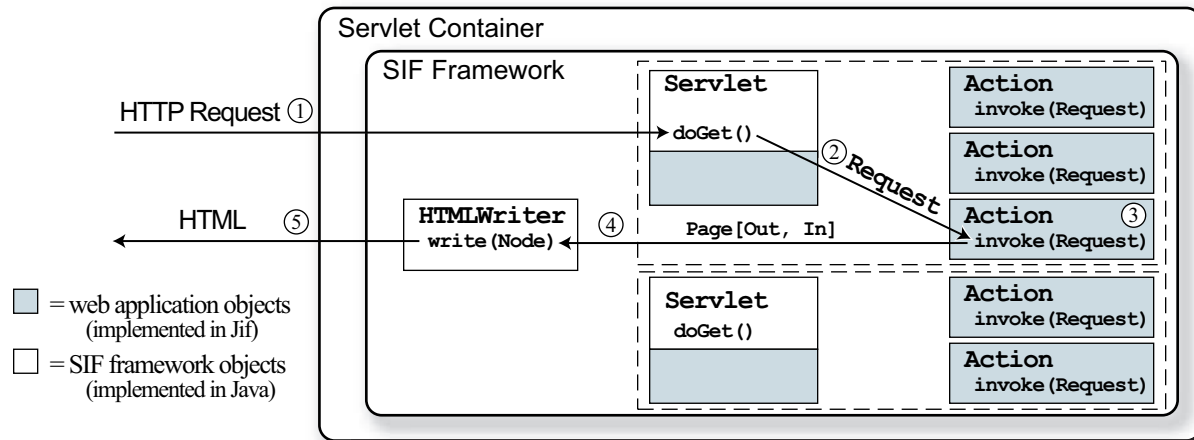


Figure 1: Handling a request in SIF.

whereas downgrading integrity reduces the set of principals considered to have influenced the information. The type system prevents unintentional downgrading by tracking the data dependencies (information flow) in the program, including *implicit flows* [8]: covert storage channels that arise from program control structure. Jif does permit information to be intentionally downgraded, but any code that does so requires the authority of all principals whose reader or writer policies are weakened or removed as a result of the downgrading.

**Timing and synchronization channels.** Jif's type system does not track information flow via timing or termination channels. These covert channels are not a serious concern if web applications are not implemented by adversaries; we assume that application programmers are not malicious. Other work (e.g., [1, 31]) has investigated checking and transforming security-typed code to remove timing channels. Termination channels (which can be regarded as an extreme timing channel) are low-bandwidth, leaking at most one bit per interaction with the web application, that is, one bit per request.

Jif was developed assuming a single-threaded execution model. However, SIF web applications are multi-threaded Jif programs, and thread synchronization can create covert timing channels that transmit information. This risk can be mitigated by configuring the web server to handle at most one concurrent request per servlet, or by isolating concurrent requests or sessions in the protection domains offered by some Java run-time systems [12, 3].

## 2.3 Design of SIF

Like the Java Servlet framework, SIF allows application code to define how client requests are handled. However, there are some structural differences that facilitate the accurate tracking of information flow. Figure 1 presents an

overview of how SIF handles a request from a web client:

1. An HTTP request is made from a web client to a servlet;
2. The HTTP request is wrapped in a Request object;
3. An appropriate Action object of the servlet is found to handle the request, and its `invoke` method called with the Request object;
4. The action's `invoke` method generates a Page object to return for the request;
5. The Page object is converted into HTML, which is returned to the client.

**Step 1: HTTP request from web client to servlet.** Web applications must extend the class `Servlet`, which is similar to the `HttpServlet` class of the Java Servlet framework. Figure 2 shows a simplified Jif signature for the `Servlet` class, as well as other key classes of SIF. The important aspects of these signatures are explained as they arise, but because of space limitations, the syntax of Jif methods and fields are not fully explained.

Web clients establish *sessions* with the servlet; sessions are tracked by the servlet container, as in the Java Servlet specification. The SIF framework creates a *session principal* for each session, which can be thought of as corresponding to the session key shared between the client and server [16], if such a key exists. The application would typically define its own user principals, which can delegate to the session principal.

**Step 2: HTTP request wrapped in a Request object.** The class `Request` is a SIF wrapper class for an HTTP request, providing restricted access to information in the request, via the `getParam` method. The restricted interface ensures that web applications are unable to circumvent the security policies on data contained in the request, as described below.

```

abstract class Servlet {
    // allows servlets to specify
    // a default action
    protected Action[req] defaultAction(Request req);

    // allows servlets to create a
    // servlet-specific SessionState object
    protected SessionState createState();

    public void setReturnPage[*:req.session](
        Request[*:req.session] req,
        label out, label in,
        Node[out,in]{*in} page)
        where {out;*in} <= {*:req.session};
}

abstract class Action {
    public abstract void
        invoke[*lbl](label{*lbl} lbl,
            Request[*lbl] req)
        where caller(req.session);
}

// base class of HTML elements
abstract class Node[label Out, label In] { }

final class Request {
    // principal representing the session
    // between client and server
    public final principal session;

    // reference to the Servlet
    public final Servlet servlet;

    // acquire a parameter value from the Request
    public String[*inp.L  $\sqcup$  inp  $\sqcup$  ( $\perp \rightarrow \perp$ ;  $T \leftarrow$  session)]
        getParam(Input inp);

    // obtain a reference to
    // the SessionState object
    public SessionState getSessionState();
}

final class Input {
    private final Nonce n;
    public final label L;
}

abstract class InputNode[label Out, label In]
    extends Node[Out,In] {
    // framework statically enforces  $Out \sqcup In \sqsubseteq inp.L$ 
    private final Input[L] inp;
}

```

Figure 2: Jif signatures for SIF classes

**Step 3: An Action is found and invoked.** Web applications implement their functionality in *actions*, which are application-defined subclasses of the SIF class `Action`. A SIF servlet may have many action objects associated with it; each action object belongs to a single servlet.

Actions can be used as the targets of forms and hyperlinks. For example, the target of a form is an action object responsible for receiving and processing the data the user submits via the form. This mechanism differs from the standard Java servlet interface, which requires the application implementor to write explicit request dispatching code (the `doGet` method). However, explicit dispatch code in the application makes precise tracking of information flow difficult, as the dispatch code is executed for all requests, even though different requests may reveal different information. By avoiding dispatch code, the action mechanism permits more precise reasoning about the information revealed by client requests to the server, as discussed further in Section 2.4.

Action objects may be *session-specific actions*, which can only ever be used by a single session, or they may be *external actions* not specific to any given session. All action objects within a given servlet have a unique identifier. For session-specific actions, the identifier is a secure nonce, automatically generated by the framework on construction of the action. For external actions, the identifier is a (human-readable) string specified by the web application. Since external actions have fixed identifiers, they may be the target of external hyperlinks, such as a hyperlink in static HTML on a different web site.

When an HTTP request is received by a servlet, the framework finds a suitable action to handle it. Typically, the HTTP request contains a parameter value specifying the unique identifier of the appropriate action; for example, forms generated by the servlet identify the action to

which the form is to be submitted. If the HTTP request does not contain an action’s unique identifier, then a default action specified by the `Servlet.defaultAction` method is used to handle the request. This default is useful for handling the first request of a new session. If the HTTP request contains an invalid action identifier (e.g., the identifier of a session-specific action of an expired or invalidated session), an error page is returned, which then redirects the user to the default action.

Actions allow web applications to maintain control over application control flow. Because session-specific actions are named with a nonce, other sessions cannot invoke them. In addition, SIF tracks the *active set* of actions for each session. An error page is returned if a request tries to invoke an action that is not active. The active set contains all external actions, and all session-specific actions that were targets of hyperlinks and forms of the last response. Thus, a client by default cannot re-submit a form by replaying its (inactive) action identifier.

Once the appropriate action object has been found, the `invoke` method is called on it with a `Request` object as an argument. The `invoke` method executes with the authority of the session principal, as shown by the `where caller(req.session)` annotation in Figure 2.

Web applications implement their functionality in the action’s `invoke` method, as Jif 3.0 code. If required, the `invoke` method can access back-end services (e.g., a database) provided that suitable Jif interfaces exist for the services. For example, web applications can access the file system since the Jif run-time library provides a Jif interface for it, which translates file system permissions into Jif security policies.

SIF web applications can provide secure web interfaces to legacy systems, by accessing the legacy systems as back-end services. The information security of

these systems is not compromised by allowing SIF applications to access them, since all accesses from Jif code must conform to the system's Jif interface.

**Step 4: The `invoke` method generates a Page object.**

An object of the class `Page` is a representation of an HTML page. SIF uses the class `Node` to represent HTML elements; the class `Page`, and other HTML elements, such as `Paragraph` and `Hyperlink`, are subclasses of `Node`. Nodes may be composed to form trees, which represent well-formed HTML code. The class `Node` is parameterized by two labels, `Out` and `In`. The `Out` label is an upper bound on the labels of information contained in the node object and its children. For example, an HTML body may contain several paragraphs, each of which contains text and hyperlinks; the `Out` parameter of each `Paragraph` node is at least as restrictive as the `Out` parameters of its child Nodes. The `In` parameter is used to bound information that may be gained by from subsequent requests originating from this page, and is discussed further in Section 2.4.

The `Action.invoke` method must generate a `Page` object, and call `Servlet.setReturnPage` with that `Page` object as an argument. The signature for `Servlet.setReturnPage` ensures that the `Out` parameter of the `Page` is at most as restrictive as the label  $\{\top \rightarrow \text{req.session}; \perp \leftarrow \perp\}$ , where `req.session` is the session principal. This label is an upper bound on all labels that permit the principal `req.session` to read information, and thus the `Page` object returned for the request can contain only information that the session principal is permitted to view. This restriction is enforced statically through the type-system, and requires no run-time examination of labels by the SIF framework. Thus, assurance is gained prior to deployment that confidential information on the server is not inadvertently released.

In addition, by requiring the application to produce `Page` objects instead of arbitrary byte sequences, SIF can ensure that each input field on a page has an appropriate security policy associated with it (see Section 2.4), and that the web application serves only well-formed HTML that does not contain possibly malicious JavaScript.

**Step 5: The Page is converted into HTML.** SIF converts the `Page` object into HTML, which is sent to the client. The `Page` object may contain hyperlinks and forms whose targets are actions of the servlet; SIF ensures that the HTML output for these hyperlinks and forms contain parameter values specifying the appropriate actions' unique identifiers; if the user follows a hyperlink or submits a form, the appropriate action is invoked.

## 2.4 Information flow over requests

The Jif compiler ensures that security policies are enforced end-to-end within a servlet, that is, from when a

request is submitted until a response is returned. However, information may flow over multiple requests within the same session, for example, by being stored in session state, or by being sent to a (well-behaved) client that returns it in the next request. SIF tracks information flow over multiple requests, to ensure that appropriate security labels are enforced on data at all times.

**Information flow through parameter values.** SIF requires each input field on a page to have an associated security label to be enforced on the input when submitted. This label is statically required to be at least as restrictive as the label of any default value for the input field, to prevent a default value from being sent back to the server with a less restrictive policy enforced on it.

SIF ensures that the submitted value of an input field has the correct label enforced on it by preventing applications from arbitrarily accessing the HTTP request's map from parameter keys to parameter values. Instead, when an input field is created in the outgoing `Page` object, an `Input` object is associated with it. An `Input` object is a pair  $(n, L)$ , where  $n$  is a freshly generated nonce, and  $L$  is the label enforced on the input value. An application can retrieve a data value from an HTTP request only by presenting the `Input` object to the `Request.getParam(Input inp)` method, which checks the nonce, and returns the submitted value with label `inp.L` enforced on it. This "closes the loop," ensuring that data sent to the client has the correct security enforced on it when the client subsequently sends it back.

SIF does not try to protect against the user copying sensitive information from the web page, and pasting into a non-sensitive input field. That is impossible in general, and the application should define labels that prevent the user from seeing information that they are not trusted to see. By keeping track of input labels, SIF prevents web applications from laundering away security policies by sending information through the client. As discussed in Section 2.5, the user can also inspect the labels on inputs to see how the application will treat the information.

The `getParam` method signature also ensures that the label  $\{\perp \rightarrow \perp; \top \leftarrow \text{session}\}$  is enforced on values submitted by the user. This label indicates that the value has been influenced by the session principal. Thus, SIF ensures that the integrity policy of any value obtained from the client correctly reflects that the client has influenced it; the Jif 3.0 compiler then ensures that this "tainted," or low-integrity, information cannot be incorrectly used as if it were "untainted," or high-integrity. This helps avoid vulnerabilities such as SQL injection, where low-integrity information is used in a high-integrity context.

**Information flow through session state.** Java servlets typically store session state in the session map of the class `javax.servlet.http.HttpSession`. How-

ever, direct access to the session map would allow SIF applications to bypass the security policies that should be enforced on values stored in the map. Instead, SIF web applications may store state in fields of session-specific actions, or in an application-defined subclass of `SessionState`. Since fields must have labels, the Jif compiler ensures that web applications honor labels associated with values stored in the state. Web applications may override the method `Servlet.createSessionState` to create an appropriate `SessionState` object; SIF ensures at run time that this method is called exactly once per session.

**Information flow through action invocation.** A subtlety of the framework is that the very act of invoking an action, by following a hyperlink or submitting a form, may reveal information to the web application. For example, if a hyperlink to some action *a* is generated if and only if some secret bit is 1, then knowing that *a* is invoked reveals the value of the secret bit.

To account for this information flow, the `Action.invoke` method takes two arguments: a label *lbl*, and a reference to the `Request` object. The label *lbl* is an upper bound on the information that may be gained by knowing which action has been invoked. This means that *lbl* must be at least as restrictive as the output information for the hyperlink or form used to invoke the action. In our example, the value of *lbl* when invoking *a* would be at least as restrictive as the label of the secret bit. In general, the value for *lbl* is the value of the `In` parameter of the `Node` that contains the link to the action; the constructors for the `Node` subclasses ensure that the parameter `In` correctly bounds the information that may be gained by knowing the node was present in the `Page` returned for the request.

The method signature for `Action.invoke` ensures that the security label *lbl* is enforced on the reference to the `Request` object (“...Request{\**lbl*} req...”.) and that *lbl* is a lower-bound for observable side-effects of the method (“invoke{\**lbl*}(...)”.) meaning that any effects of the method (such as assignments to fields) must be observable only at security levels bounded below by *lbl*. These restrictions ensure that SIF correctly tracks the information that may be gained by knowing which actions were available for the user to invoke.

## 2.5 Deploying SIF web applications

SIF web applications may be deployed on standard Java Servlet containers, such as Apache Tomcat, and thus may be used in a multi-tier architecture wherever Java servlets are used. The SIF and Jif run-time libraries must be available on the class path, but deployment of SIF web applications is otherwise similar to deployment of ordinary Java servlets. The deployer of a SIF web application

is free to specify configuration information in the application’s deployment descriptor (the `web.xml` file). For example, the deployer may require all connections to use SSL, thus protecting the confidentiality and integrity of information in transit between client and server. Additionally, there are several SIF-specific options that a deployer may specify in the deployment descriptor.

**Cascading style sheets.** SIF applications must use the `Node` subclasses to generate responses to requests, which allows them to generate only well-formed HTML. To allow flexibility in presentation details such as colors and font attributes, SIF permits the deployment descriptor to specify a cascading style sheet (CSS) to use in the presentation of all HTML pages generated by the application; SIF adds this URL in the head of all generated HTML pages. `Node` objects can specify a `class` attribute, allowing style sheets to provide almost arbitrary formatting. While this allows great flexibility, care must be taken that the CSS does not contain misleading formatting. For example, inappropriate formatting might lead a user to enter sensitive information into a non-sensitive input field, such as a social security number into an address field. The deployer should review the CSS before deploying the application.

**JavaScript.** Dynamically generated JavaScript can provide rich user interfaces, but introduces new possibilities for security violations and covert channels. SIF does not allow web applications to send dynamic JavaScript to the client. However, as with CSSs, SIF allows deployment descriptors to specify a URL containing (static) JavaScript code to be included on all generated HTML pages. Explicit inclusion of JavaScript permits easy review by the deployer. Ideally, SIF should automatically check included JavaScript code (or perhaps an extension of JavaScript with information-flow control); we leave this to future work.

**Policy visualization.** User awareness of security policies is an important aspect of secure systems. Since SIF tracks the policies of information sent to the user, SIF can augment the user interface to inform the user of the security policies of data they view and supply. Provided the user trusts the interface (see Section 2.1), this helps prevent, for instance, a user from inappropriately copying sensitive information from the browser into an email, or from following an untrusted hyperlink.

Web applications may opt to allow SIF to automatically color-code information sent to the client, based on policy annotations. When the user presses a hotkey combination, JavaScript code recolors the page elements to reflect their confidentiality, varying from red (highly confidential) to green (low confidentiality). Both displayed information and inputs are colored appropriately. An additional hotkey colors the page based on the integrity

policies of information. A third hotkey shows a legend of colors and corresponding labels so the user can identify the precise security policy for each page element.

### 3 Language extensions

Web applications have diverse, complicated, and dynamic security requirements. For example, web applications display a plethora of authentication schemes, including various password schemes, password recovery schemes, biometrics, and CAPTCHAs to identify human users. Web applications often enforce dynamic security policies, such as allowing users to specify who may view and update their information. Moreover, the security environment of a web application is dynamic: new users are being created, users are starting and ending sessions, and authenticating themselves.

In order both to accommodate diverse, complicated, and dynamic security requirements, and to provide assurance that these requirements are met, we have produced Jif 3.0, a new version of Jif. Section 2.2 describes the previous version of Jif; this section presents new features that support dynamic security requirements: integration of information flow with application-defined authentication and authorization, and improved ability to reason about and compute with dynamic security labels and principals.

Care was needed in the design and implementation of these language extensions, since there is always a tension in language-based security between expressiveness and security. In particular, the new dynamic security mechanisms in Jif 3.0 create new information channels, complicating static analysis of information flow. Importantly, Jif 3.0 tracks these channels to prevent their misuse.

#### 3.1 Application-specific principals

Principals are entities with security concerns. Applications may choose which entities to model as principals. Principals in Jif are represented at run time, and thus can be used as values by programs during execution. Jif gives run-time principals the primitive type `principal`. Jif 3.0 introduces an open-ended mechanism that allows applications great flexibility in defining and implementing their own principals.

Applications may implement the Jif 3.0 interface `jif.lang.Principal`, shown in simplified form in Figure 3. Any object that implements the `Principal` interface is a principal; it can be cast to the primitive type `principal`, and used just as any other principal. The `Principal` interface provides methods for principals to delegate their authority and to define authentication.

Delegation is crucial. For example, user principals must be able to delegate their authority to session principals, so that requests from users can be executed with

```
interface Principal {
    String name();
    // does this principal delegate authority to q?
    boolean delegatesTo(principal q);
    // is this principal prepared to authorize the
    // closure c, given proof object authPrf?
    boolean isAuthorized(Object authPrf,
                        Closure[this] c);
    // methods to guide search for acts-for proofs
    ActsForProof findProofUpTo(Principal p);
    ActsForProof findProofDownTo(Principal q);
}
interface Closure[principal P] authority(P) {
    // authority of P is required to
    // invoke a Closure
    Object invoke() where caller(P);
}
```

Figure 3: Signatures for application-specific principals

their authority. The method call `p.delegatesTo(q)` returns true if and only if principal `p` delegates its authority to principal `q`. The implementation of a principal's `delegatesTo` method is the sole determiner of whether its authority is delegated. An *acts-for proof* is a sequence of principals  $p_1, \dots, p_n$ , such that each  $p_i$  delegates its authority to  $p_{i+1}$ , and is thus a proof that  $p_n$  can act for  $p_1$ . Acts-for proofs are found using the methods `findProofUpTo` and `findProofDownTo` on the `Principal` interface, allowing an application to efficiently guide a proof search. Once an acts-for proof is found, it is verified using `delegatesTo`, cleanly separating proof search from proof verification.

The authority of principals is required for certain operations. For example, the authority of the principal *Alice* is required to downgrade information labeled  $\{Alice \rightarrow Bob ; \top \leftarrow \top\}$  to the label  $\{Alice \rightarrow Bob, Chuck ; \top \leftarrow \top\}$  since a policy owned by *Alice* is weakened. The authority of principals whose identity is known at compile time may be obtained by these principals approving the code that exercises their authority. However, for dynamic principals, whose identity is not known at compile time, a different mechanism is required. We have extended Jif with a mechanism for dynamically authorizing closures.

An *authorization closure* is an implementation of the interface `jif.lang.Closure`, shown in Figure 3. The `Closure` interface has a single method `invoke`, and is parameterized on a principal `P`. The `invoke` method can only be called by code that possesses the authority of principal `P`, as indicated by the annotation `where caller(P)`. Code that does not have the authority of principal `P` can request the Jif run-time system to execute a closure for `P`; the run-time system will do so only if `P` authorizes the closure.

The `Principal` interface provides a method for authorizing closures, `isAuthorized`. It takes two arguments: a `Closure` object instantiated with the principal represented by the `this` object, and an application-specific proof of authentication and/or authorization.

For example, the proof might be a password, a checkable proof that the closure satisfies certain safety requirements, or a collection of certificates or capabilities. The application-specific implementation of the `isAuthorized` method examines the closure and the proof object, and returns `true` if the principal grants its authority to the closure.

The `Principal` interface and authorization closures provide a flexible mechanism for web applications to implement their own authentication and authorization mechanisms. For example, in the case studies of Section 4, closures are used to obtain the authority of application users after they have authenticated themselves with a password. Other implementations of principals are free to choose other authentication and authorization mechanisms, such as delegating the authorization decision to a XACML service. Dynamic authorization tests introduce new information flows that are tracked using Jif's security-type system. To prevent the usurpation of a principal's authority, the Jif run-time library cannot execute a closure unless appropriately authorized.

Legacy systems may have their own abstractions for users, authentication, and authorization. Application-specific principals allow legacy-system security abstractions to be integrated with web applications. For example, when integrating with a database with access controls, database users can be represented by suitable implementations of the `Principal` interface; web applications can then execute queries under the authority of specific database users, rather than executing all queries using a distinguished web server user.

## 3.2 Dynamic labels and principals

Jif can represent labels at run time, using the primitive type `label` for run-time label values. Following work by Zheng and Myers [42], Jif 3.0's type system has been extended with more precise reasoning about run-time labels and principals. It is now possible for the label of a value (or a principal named in a label) to be located via a *final access path expression*. A final access path expression is an expression of the form  $r.f_1 \dots f_n$ , where  $r$  is either a final local variable (including final method arguments), or the expression `this`, and each  $f_i$  is an access to a final field. For example, in Figure 2, the signature for the method `Request.getParam(Input inp)` indicates that the return value has the label `inp.L` enforced on it. Therefore, the Jif 3.0 compiler can determine that the label of the result of the `getParam` method is found in the object `inp`. The additional precision of Jif 3.0 is needed to capture this relationship.

This additional precision allows SIF web applications to express and enforce dynamic security requirements, such as user-specified security policies. SIF web applications can also statically control information received

from the currently authenticated user, whose identity is unknown at compile time.

The use of dynamic labels and principals introduces new information flows, because which label is enforced on information may itself reveal information. Jif 3.0's type system tracks such flows, and prevents dynamic labels and principals from introducing covert channels.

## 3.3 Caching dynamic tests

To allow efficient dynamic tests of label and principal relations, the Jif 3.0 runtime system caches the results of label and principal tests. Separate caches are maintained for positive and negative results of acts-for and label tests. Care must be taken that the use of caches does not introduce unsoundness. When a principal delegation is added, the negative acts-for and label caches are cleared, as the new delegation may now enable new relationships. When a principal delegation is removed, entries in the positive acts-for and label caches that depend upon that delegation are removed, as the relationship may no longer hold.

When principals add or remove delegations, they should notify the Jif 3.0 runtime system, which updates the caches appropriately. Although an incorrectly or maliciously implemented principal  $p$  may fail to notify the runtime system, lack of notification can hurt only the principal  $p$ , since  $p$  (and only  $p$ ) determines to whom its authority is delegated.

## 4 Case studies

Using SIF, we have designed and implemented two web applications. The first is a cross-domain information sharing system that permits multiple users to exchange messages. The second is a multi-user calendar application that lets users create, edit, and view events.

This section describes the key functionality of these applications, their information security requirements, and how we reflected these requirements in the implementations. Real applications must release information, reducing its confidentiality. In SIF, this is implemented by *downgrading* to a lower security label. We discuss and categorize downgrades that occur in the applications. Based on our experience, we make some observations about programming with information-flow control.

### 4.1 Application descriptions

**Cross-domain information sharing (CDIS).** CDIS applications involve exchange of information between different entities with varying levels of trust between them. For example, organizational policy may require the approval of a manager to share information between members of certain departments. Many CDIS systems provide an automatic process; for example, they determine

what approval is needed, and delay information delivery until approval is obtained.

We have designed and implemented a prototype CDIS system. The interface is similar to a web-based email application. The application allows users to log in and compose messages to each other. A message may require review and approval by other users before it is available to its recipients. The review process is driven by a set of system-wide mandatory rules: each rule specifies for a unique sender-recipient pair which users need to review and approve messages. Once all appropriate reviewers have approved a message, it appears in the recipient's inbox. Each user also has a "review inbox," for messages requiring their approval or rejection. In this prototype, all messages are held centrally on the web server; a full implementation would be integrated with an SMTP server.

**Calendar.** We have also implemented a multi-user calendar system. Authenticated users may create, edit, and view events. Events have a time, title, list of attendees, and description. Events are controlled by expressive security policies, customizable by application users. A user can edit an event only if the user acts for the creator of the event (recall that the *acts-for* relation is reflexive). A user may view the details of an event (title, attendees, and description) if the user acts for either the creator or an attendee. An event may specify a list of additional users who are permitted to view the time of the event—to view an event, a user must act for the creator, for an attendee, or for a user on this list.

A user's calendar is defined to be the set of all events for which the user is either the creator or an attendee. When a user  $u$  views another user  $v$ 's calendar,  $u$  will see only the subset of events on  $v$ 's calendar for which  $u$  is permitted to see the details or time. If the user is permitted to view the time, but not the details of an event, the event is shown as "Busy."

**Measurements.** Measurements of the applications' code are given in Figure 4, including non-blank non-comment lines of code, lines with label annotations, and the number of declassify and endorse annotations, which indicate intentional downgrading of information (see Section 4.3).

Performance tests indicate that the overhead due to the SIF framework is modest. We compared the calendar case study application to a Java servlet we implemented with similar functionality, using the same backend database; the Java servlet does not offer the security assurances of the SIF servlet. Tests were performed using Apache Tomcat 5.5 in Redhat Linux, kernel version 2.6.17, running on a dual-core 2.2GHz Opteron processor with 3GB of memory. As the number of concurrent sessions varies between 1 and 245, the SIF servlet exhibits at most a 29% reduction in requests processed per second, showing that SIF does not dramatically af-

fect scalability. At peak throughput, the Java servlet processes 2010 requests per second, compared with 1503 for the SIF servlet. Of the server processing time for a request to the SIF servlet, about 17% is spent rendering the Page object into HTML, and about 9% is spent performing dynamic label and principal tests.

## 4.2 Implementing security requirements

Many of the security requirements of both applications can be expressed using Jif's security mechanisms, including dynamic principals and security labels, and thus automatically enforced by Jif and SIF's static and run-time mechanisms. Other security requirements are enforced programmatically.

**Principals.** Users of the applications are application-specific principals (see Section 3.1). We factored out much functionality from both applications relating to user management, such as selecting users and logging on and off. The sharing of code across both case studies shows that SIF permits the design and implementation of reusable components. Figure 4 also shows measurements of the reusable user library.

The login process works as follows: a user and password are specified on the login screen, and if the password is correct, the authority of the user is dynamically obtained via a closure; the closure is used to delegate the user's authority to the session principal, who can then act on behalf of the now logged-in user.

In addition to user principals, the two applications define principals `CDISApp` and `CalApp`, representing the applications themselves. These model the security of sensitive information that is not owned by any one user, such as the set of application users. This information is labeled  $\{p \rightarrow \top ; p \leftarrow \top\}$ , where  $p$  is one of `CDISApp` or `CalApp`, and relevant portions are downgraded for use as needed. In particular, information in the database has this label. Since all information sent to and from the database (including data used in SQL queries) must have this label, the authority of the application principal (`CDISApp` or `CalApp`) is required to endorse information sent to the database and to declassify information received from it. This provides a form of access control, ensuring that only code authorized by the application principal is able to access the database. The need to explicitly endorse data used in SQL queries also helps to prevent SQL injection attacks, by making the programmer aware of exactly what information may be used in SQL queries.

**Dynamic security labels.** The security labels of Jif 3.0 are expressive enough to capture the case studies' information-sharing requirements. In particular, we are able to model the confidentiality and review requirements for CDIS messages by enforcing appropriate labels on the messages. For instance, suppose sender  $s$

	Lines	Annotated Lines	Downgrade Annotations	Functional downgrades			
				Access control	Imprecision	Application	Total
CDIS	1325	277	76	11	0	3	14
Calendar	1779	443	73	12	0	5	17
User	925	283	31	3	1	4	8

Figure 4: Summary of case studies.

is sending a message to recipient  $t$ . The confidentiality policy  $s \rightarrow t$  would allow both  $s$  and  $t$  to read the message. However, before  $t$  is permitted to read the message, it may need to be reviewed. Suppose reviewers  $r_1, r_2, \dots, r_n$  must review all messages sent from  $s$  to  $t$ . When  $s$  composes the message, it initially has the following confidentiality policy:  $(s \rightarrow t, r_1, \dots, r_n) \sqcup (r_1 \rightarrow r_1, \dots, r_n) \sqcup \dots \sqcup (r_n \rightarrow r_1, \dots, r_n)$ . In this policy,  $s$  permits  $t$  and all reviewers to read the message, and each reviewer permits all other reviewers to read the message. This label allows the message to be read by each reviewer, but prevents  $t$  from reading it. As each reviewer reviews and approves the message, their authority is used to remove their reader policy from the confidentiality policy using *declassify* annotations. Eventually the message is declassified to the policy  $s \rightarrow t, r_1, \dots, r_n$ , which permits  $t$  to read it.

The calendar application also enforces user-defined security requirements by labeling information with appropriate dynamic labels. Event details have the confidentiality policy  $c \rightarrow a_1, \dots, a_n$  enforced on them, where  $c$  is the creator of the event and  $a_1, \dots, a_n$  are the event attendees. The time of an event has confidentiality policy  $c \rightarrow a_1, \dots, a_n \sqcap c \rightarrow t_1, \dots, t_m$ , where  $t_1, \dots, t_m$  are the users explicitly given permission by  $c$  to view the event time. Event labels ensure that times and details flow only to users permitted to see them; run-time label tests are used to determine which events a user can see.

### 4.3 Downgrading

Jif prevents the unintentional downgrading of information. However, most applications that handle sensitive information, including the case study applications, need to downgrade information as part of their functionality. Jif provides a mechanism for deliberate downgrading of information: *selective declassification* [22, 26] is a form of access control, requiring the authorization of the owners of all policies weakened or removed by a downgrade. Authorization can be acquired statically if the owner of a policy is known at compile time; or authorization can be acquired at run time through a closure (see Section 3).

Jif 3.0 programs must also satisfy typing rules to enforce *robust declassification* [40, 23, 5]. In the context of Jif, robustness ensures that no principal  $p$  (including attackers) is able to influence either *what* information is released to  $p$  (a *laundering attack*), or *whether* to release information to  $p$ . For a web application, robustness im-

plies that users are unable to cause the incorrect release of information. Selective declassification and robust declassification are orthogonal, providing different guarantees regarding the downgrading of information.

In Jif programs, downgrading is marked by explicit program annotations. A *declassify* annotation allows confidentiality to be downgraded, whereas an *endorse* annotation downgrades integrity.

Downgrading annotations are typically clustered together in code, with several annotations needed to accomplish a single “functional downgrade.” For example, declassifying a data structure requires declassification of each field of the structure [2]. The two applications had a combined total of 39 functional downgrades, with an average of 4.6 annotations per functional downgrade.

Figure 4 shows a more detailed breakdown of the use of downgrading in each case study. (Details of each downgrade appear in Appendix A.) We found that downgrading could be divided into three broad categories: access control, imprecision, and application requirements.

The first category is downgrades associated with discretionary access control. Discretionary access control is used as a mechanism to mediate information release between different application components; any information release requires explicit downgrading. For example, in the calendar application, the set of all events has the label  $\{\text{CalApp} \rightarrow \top ; \text{CalApp} \leftarrow \top\}$ ; thus, downgrading is required both to extract events to display to the user, and to update events edited by the user; the authority of *CalApp* is required for these downgrades, and thus the downgrades serve as a form of discretionary access control to the event set. The choice of the label  $\{\text{CalApp} \rightarrow \top ; \text{CalApp} \leftarrow \top\}$  for the event set necessitates these downgrades; using other labels may result in fewer downgrades, but without the benefits of this discretionary access control.

Imprecision is another reason for downgrading: sometimes the programmer can reason more precisely than the compiler about security labels and information flows. For example, suppose a method is always called with a non-null argument: Jif 3.0 has no ability to express this precondition, and conservatively assumes that accessing the argument may result in a *NullPointerException*. Since the exception may reveal information, a spurious information flow is introduced, which may require explicit downgrading later. Few downgrades fall into this category, giving confidence that Jif 3.0 is sufficiently ex-

pressive. Some imprecision could be removed entirely by extending the compiler to accept and reason about additional annotations, as in JML [17].

Security requirements of the application provide the third category of downgrade reasons. These downgrades are inherent in the application, and cannot and should not be avoided. For example, in the calendar application, when users are added to the list of event attendees, more users are able to see the details of the event, an information release that requires explicit downgrading.

#### 4.4 Programming with information flow

During the case studies' development, we obtained several insights into the design and implementation of applications with information flow control.

**Abstractions and information flow.** Information flow analysis tends to reveal details of computations occurring behind encapsulation boundaries, making it important to design abstractions carefully. Unless sufficient care is taken during design, abstractions will need to be modified during implementation. For example, we sometimes needed to change a method's signature several times, both while implementing the method body (and discovering flows we hadn't considered during design), and while calling the method in various contexts (as method invocation may reveal information to the callee, which we hadn't considered when designing the signature).

**Coding idioms.** We found that certain coding idioms simplified reasoning about information flow, by putting code in a form that either allowed the programmer to better understand it, or allowed Jif's type system to reason more precisely about it. As a simple example, consider the following (almost) equivalent code-snippets for assigning the result of method call `o.m()` to `x`, followed by an assignment to `y`:

1. `x = o.m(); y = 42;`
2. `if (o != null) { x = o.m(); } y = 42;`

The first snippet throws a `NullPointerException` if `o` is null, and thus information about the value of `o` flows to `x`, and also to `y` (since the assignment to `y` is executed only in the absence of an exception). The information flow to `y` is subtle, and a common trap for new Jif programmers. In the second snippet, no exception can be thrown (the compiler detects this with a data-flow analysis), and so information about `o` does not flow to `y`. This snippet avoids the subtle implicit flow to `y`. More generally, making implicit information flow explicit simplifies reasoning about information flow.

**Declarative security policies.** Many of the case studies' security requirements were expressed using Jif labels. SIF and the Jif compiler ensure that these labels (and thus the security requirements) are enforced end-to-end. In general, Jif's declarative security policies can re-

lieve the programmer of enforcing security requirements programmatically, and give greater assurance that the requirements are met. This argues for even greater expressiveness in security policies, to allow more application security requirements to be captured, and to verify that programs enforce these requirements.

## 5 Related work

The most closely related work is Li and Zdancewic's [18], which proposes a security-typed PHP-like scripting language to address information-flow control in web applications. Their system has not been implemented. It assumes a strongly-typed database interface, and, like SIF, ensures that applications respect the confidentiality and integrity policies on data sent to and from the database. Their security policies can express *what* information may be downgraded; in contrast, the decentralized label model used in Jif specifies *who* needs to authorize downgrading. In a multi-user web application with mutually distrusting users, the concept of *who* a session or process is executing on behalf of is crucial to security. We believe that practical information-flow control will ultimately need to specify multiple aspects of downgrading [29]; extending the decentralized label model to reason about other downgrading aspects is ongoing work.

Huang et al. [14], Xie and Aiken [37], and Jovanovic et al. [15] all present frameworks for statically analyzing information flow in PHP web applications. Xie and Aiken, and Jovanovic et al. track information integrity using a dataflow analysis, while Huang et al. extend PHP's type system with type state. Livshits and Lam [19] use a precise static analysis to detect vulnerabilities in Java web applications. Each of these frameworks has found previously unknown bugs in web applications. Xu et al. [38], Halfond and Orso [11] and Nguyen-Tuong et al. [25] use dynamic information-flow control to prevent attacks in web applications. All of these approaches use a simple notion of integrity: information is either *tainted* or *untainted*. While this suffices to detect and prevent certain web application vulnerabilities, such as SQL injection, it is insufficient for modeling more complex, application-level integrity requirements that arise in applications with multiple mutually distrusting principals. Also, they do not address confidentiality information flows, and thus do not control the release of sensitive server-side information to web clients.

Xu et al. [39] propose a framework for analyzing and dynamically enforcing client privacy requirements in web services. They focus on web service composition, assuming that individual services correctly enforce policies. Their policies do not appear suitable for reasoning about the security of mutually distrusting users. Otherwise, this work is complementary, as we provide

assurance that web applications enforce security policies.

While there has been much recent work on language-based information flow (see [28, 29] for recent surveys), comparatively little has focused on creating real systems with information flow security, or on languages and techniques to enable this. No prior work has built real applications that enforce both confidentiality and integrity policies while dealing securely with their interactions.

The most realistic prior application experience is that of Hicks et al. [13], who use an earlier version of Jif to implement a secure CDIS email client, JPmail. Although there are similarities between JPmail and the CDIS mail application described here, SIF is a more convincing demonstration of information flow control in three ways. First, SIF is a reusable application framework, not just a single application. Second, SIF applications enforce integrity, not just confidentiality, and they ensure that declassification is robust [5]. Third, SIF applications can dynamically extend the space of principals and labels and define their own authentication mechanisms; JPmail relies on mechanisms for principal management and authentication that lie outside the scope of the application.

Askarov and Sabelfeld [2] use Jif to implement cryptographic protocols for mental poker. They identify several useful idioms for (and difficulties with) writing Jif code; recent extensions to Jif should assuage many of the difficulties.

Praxis High Integrity System's language SPARK [4] is based on a subset of Ada, and adds information-flow analysis. SPARK checks simple dependencies within procedures. FlowCaml [27] extends the functional language OCaml with information-flow security types. Like SPARK, it does not support features needed for real applications: downgrading, dynamic labels, and dynamic and application-defined principals.

Asbestos [9], Histar [41], and SELinux [20] are operating systems that track information flow for confidentiality and integrity. To varying degrees, they provide flexible security labels and application-defined principals. However, these systems are coarse-grained, tracking information flow only between processes. Information flow is controlled only dynamically, which is imprecise, and creates additional information flows from runtime label checking. By contrast, Jif checks information flow mostly statically, at the granularity of program variables, providing increased precision and greater assurance that a program is secure prior to deployment. Asbestos has a web server that allows web applications to isolate users' data from one another, using one process per user. All downgrades are performed by trusted processes. Unlike Jif, this granularity of information flow tracking does not permit different security policies for different data owned by a single user.

Tse and Zdancewic [35] present a monadic type sys-

tem for reasoning about dynamic principals, and certificates for authority delegation and downgrading. Jif 3.0's dependent type system for dynamic labels and principals allows similar reasoning. Tse and Zdancewic assume that certificates are contained in the external environment, and do not provide a mechanism to dynamically create them. Closures in Jif 3.0 can be dynamically authorized, and may perform arbitrary computation, whereas Tse and Zdancewic's certificates permit only authority delegation and downgrading.

Swamy et al. [32] consider dynamic policy updates, and introduce a transactional mechanism to prevent *unintentional transitive flows* that may arise from policy updates. In Jif, policies are updated dynamically by adding and removing principal delegations, and unintentional transitive flows may occur. Their techniques are complementary to our work, and should be applicable to Jif to stop these flows.

## 6 Conclusion

We have designed and implemented Servlet Information Flow (SIF), a novel framework for building high-assurance web applications. Extending the Java Servlet framework, SIF addresses trust issues in web applications, moving trust out of web applications and into SIF and the Jif compiler.

SIF web applications are written entirely in the Jif 3.0 programming language. At compile time, applications are checked to see if they respect the confidentiality and integrity of information held on the server: confidential information is not released inappropriately to clients, and low-integrity information from clients is not used in high-integrity contexts. SIF tracks information flow both within the handling of a single request, and over multiple requests—it closes the loop of information flow between client and server.

Jif 3.0 extends Jif in several ways to make web applications possible. It adds sophisticated dynamic mechanisms for access control, authentication, delegation, and principal management, and shows how to integrate these features securely with language-based, largely static, information-flow control.

We have used SIF to implement two applications with interesting information security requirements. These web applications are among the first to statically enforce strong and expressive confidentiality and integrity policies. Many of the applications' security requirements were expressible as security labels, and are thus enforced by the Jif 3.0 compiler.

As language-based information-flow control becomes more mature, and information-flow tools become more useful and robust, we expect the task of writing and understanding programs with information-flow control to become easier. This work makes an important step to-

wards wider use of information-flow control by providing a framework in which useful applications can be designed, implemented, and deployed. The Jif 3.0 compiler and run-time system and the SIF framework are all publicly available.

## Acknowledgments

We thank Nate Nystrom, Lantian Zheng, Xin Qi, and Jed Liu for useful suggestions. The research was supported in part by NSF awards 0430161 and 0627649 and by an Alfred P. Sloan Research Fellowship, and in part by TRUST (The Team for Research in Ubiquitous Secure Technology) and AF-TRUST (Air Force Team for Research in Ubiquitous Secure Technology for GIG/NCES), which receive support from the NSF (award 0424422) and from AFOSR (FA9550-06-1-0244), Cisco, British Telecom, ESCHER, HP, IBM, iCAST, Intel, Microsoft, ORNL, Pirelli, Qualcomm, Sun, Symantec, Telecom Italia and United Technologies.

## References

- [1] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, January 2000.
- [2] Aslan Askarov and Andrei Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. 10th European Symposium on Research in Computer Security (ESORICS)*, September 2005.
- [3] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proc. 4th Symposium on Operating Systems Design and Implementation*. USENIX, October 2000.
- [4] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, April 2003.
- [5] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *Proc. 19th IEEE Computer Security Foundations Workshop*, July 2006.
- [6] Lap chung Lam and Tzi cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proc. 22st Annual Computer Security Applications Conference (ACSAC 2006)*, December 2006.
- [7] Danny Coward and Yutaka Yoshida. Java servlet specification, version 2.4, November 2003. JSR-000154.
- [8] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7), July 1977.
- [9] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proc. 20th ACM Symp. on Operating System Principles (SOSP)*, October 2005.
- [10] T. Garfinkel, B. Baff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine based platform for trusted computing. In *Proc. 19th ACM Symp. on Operating System Principles (SOSP)*, 2003.
- [11] W. Halfond and A. Orso. AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks. In *Proc. International Conference on Automated Software Engineering (ASE'05)*, pages 174–183, November 2005.
- [12] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *Proc. 1998 USENIX Annual Technical Conference*, June 1998.
- [13] Boniface Hicks, Kiyan Ahmadi-zadeh, and Patrick McDaniel. Understanding practical application development in security-typed languages. In *22nd Annual Computer Security Applications Conference (ACSAC)*, Miami, FL, December 2006.
- [14] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proc. 13th International Conference on World Wide Web*. ACM Press, 2004.
- [15] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proc. IEEE Symposium on Security and Privacy*, May 2006.
- [16] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, October 1991. *Operating System Review*, 253(5).
- [17] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, pages 105–106, Minneapolis, Minnesota, 2000.
- [18] Peng Li and Steve Zdancewic. Practical information-flow control in web-based information systems. In *Proc. 18th IEEE Computer Security Foundations Workshop*, 2005.
- [19] V. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proc. 14th USENIX Security Symposium (USENIX'05)*, pages 271–286, August 2005.
- [20] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.
- [21] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, January 1999.
- [22] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4), October 2000.
- [23] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *Proc. 17th IEEE Computer Security Foundations Workshop*, June 2004.
- [24] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. Software release, at <http://www.cs.cornell.edu/jif>, July 2001–.
- [25] A. Nguyen-Tuong, S. Guarneri, D. Greene, and D. Evans. Automatically hardening web applications using precise tainting. In *Proc. 20th International Information Security Conference*, pages 372–382, May 2005.
- [26] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. 5nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.
- [27] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, 2002.
- [28] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.
- [29] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proc. 18th IEEE Computer Security Foundations Workshop*, June 2005.
- [30] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proc. 20th ACM Symp. on Operating System Principles (SOSP)*, pages 1–15, October 2005.
- [31] Geoffrey Smith. A new type system for secure information flow. In *Proc. 14th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, June 2001.
- [32] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *Proc. 19th IEEE Computer Security Foundations Workshop*, pages 202–216, July 2006.
- [33] Symantec internet security threat report, volume IX. Symantec Corporation, March 2006.
- [34] Trusted Computing Group. *TCG TPM Specification Version 1.2*

Revision 94, March 2006.

- [35] Stephen Tse and Steve Zdancewic. Designing a security-typed language with certificate-based declassification. In *Proc. 14th European Symposium on Programming*, 2005.
- [36] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proc. 7th International Joint Conference on the Theory and Practice of Software Development*, 1997.
- [37] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. 15th USENIX Security Conference*, July 2006.
- [38] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, August 2006.
- [39] Wei Xu, V.N. Venkatakrishnan, R. Sekar, and I.V. Ramakrishnan. A framework for building privacy-conscious composite web services. In *4th IEEE International Conference on Web Services (ICWS'06)*, September 2006.
- [40] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.
- [41] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, November 2006.
- [42] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Proc. 2nd Workshop on Formal Aspects in Security and Trust, IFIP TC1 WG1.7*. Springer, August 2004.

## A Downgrading in case studies

These tables describe the case studies' functional downgrades.

### CDIS application

Description	Category
<b>Error composing message.</b> If an error is made when composing a message (e.g., leaving Subject field empty), the user is sent back to message composition. Downgrading this information flow reveals very little about the message data.	Application
<b>Message approval.</b> When a reviewer approves a message, he downgrades his confidentiality restriction. Once all reviewers have approved the message, the recipient may view it.	Application
<b>Database access.</b> Access to the database is done with the authority of the principal CDISApp. There are 11 functional downgrades for database accesses, releasing info from CDISApp to the user.	Access control
<b>Delegation to CDISRoot.</b> All users delegate authority to a root user for the CDIS application, CDISRoot, to perform operations that affect all users. This delegation requires user endorsement.	Application

### User library

Description	Category
<b>Unsuccessful login.</b> When user enters a password on the login page, he learns if the password was correct. If incorrect, the user is returned to the login page with an error message. This information release about the password is acceptable.	Application
<b>Successful login.</b> When the user logs in successfully, he learns that the password was correct. This information flow is secure.	Application
<b>Delegation to session principal.</b> When the user logs in, he delegates authority to the session principal, using a closure. The decision to authorize the delegation closure must be declassified.	Application

<b>Delegation to session principal.</b> Delegating authority from a newly logged in user to the session principal requires the trust of the user, and thus an endorsement.	Application
<b>Retrieving users from the database.</b> When selecting one or more users, info must be retrieved from the database, and returned to the caller of the Select User(s) page. This transfer requires a total of 3 functional downgrades during user selection.	Access control
<b>Error selecting user(s).</b> A user making an error on the Select User(s) page (e.g., no user id entered) is returned to the Select User(s) page. As this page is a reusable component, its label is set conservatively. A declassification is needed for the error message, from the conservative label to the actual label used for a given page invocation.	Imprecision

### Calendar application

Description	Category
<b>Update session state with date to display.</b> The display date must be trusted by the session principal. The date input by the user is trusted by the user, but must be endorsed by the session principal before it's stored in session state.	Access control
<b>Update session state with which user's calendar to display.</b> Similarly, the user selects a user's calendar to display. This downgrade ensures that the session principal authority is required to update session state.	Access control
<b>Fresh id for new event.</b> A new event requires a fresh unique id. The unique id may act as a covert channel, revealing info about the order in which events are created. Since ids are generated randomly, downgrading the fresh id is secure.	Application
<b>Update and retrieve info from database.</b> When info needs to be updated in the database (e.g., edit an event) or retrieved (e.g., fetch user details, or events) information must be transferred between the current user and the application principal CalApp. There are 10 such functional downgrades, for different database accesses.	Access control
<b>Go to View/Edit Event page.</b> An event's name is displayed as a hyperlink to the View Event or Edit Event page (depending on user's permissions). Since the link contains the event's name, the info gained by invoking View/Edit Event action is at least as restrictive as the event detail's label. This reveals little about which event is being viewed/edited.	Application
<b>Error editing event.</b> The user who makes an error editing an event (e.g., end time before start) is sent back to the Edit Event page. Like the "Go to View/Edit Event" downgrade, this reveals little about the data input.	Application
<b>Changing attendees or viewers of an event.</b> When the user edits an event and changes the attendees or viewers of an event, the labels to enforce on the event time and details change. This requires a downgrade.	Application
<b>Delegation to CalRoot.</b> All users delegate their authority to a root user for the calendar application, CalRoot, whose authority is needed to perform operations that affect all users. This requires an endorsement from each user.	Application

# Combating Click Fraud via Premium Clicks

Ari Juels

*RSA Laboratories*<sup>1</sup>  
ajuels@rsa.com

Sid Stamm

*Indiana University, Bloomington*  
sstamm@indiana.edu

Markus Jakobsson

*Indiana University, Bloomington and RavenWhite Inc.*  
markus@indiana.edu

## Abstract

We propose new techniques to combat the problem of click fraud in pay-per-click (PPC) systems. Rather than adopting the common approach of filtering out seemingly fraudulent clicks, we consider instead an affirmative approach that only accepts legitimate clicks, namely those validated through client authentication. Our system supports a new advertising model in which “premium” validated clicks assume higher value than ordinary clicks of more uncertain authenticity. Click validation in our system relies upon sites sharing evidence of the legitimacy of users (distinguishing them from bots, scripts, or fraudsters). As cross-site user tracking raises privacy concerns among many users, we propose ways to make the process of authentication anonymous. Our premium-click scheme is transparent to users. It requires no client-side changes and imposes minimal overhead on participating Web sites.

**Key words:** authentication, click-fraud

## 1 Introduction

*Pay-per-click* (PPC) metering is a popular payment model for advertising on the Internet. The model involves an *advertiser* who contracts with a specialized entity, which we refer to as a *syndicator*, to distribute textual or graphical banner advertisements to *publishers* of content. These banner ads point to the advertiser’s Web site: When a user clicks on the banner ad on the publisher’s webpage, she is directed to the site to which it points. Search engines such as Google and Yahoo are the most popular syndicators, and create the largest portion of pay-per-click traffic on the Internet today. These sites display advertisements on their own search pages in response to the search terms entered by users and charge advertisers for clicks on these links (thereby acting as their own publishers) or, increasingly, outsource advertisements to third-party publishers. Advertisers pay syndicators per referral, and the syndicators pass on a portion of the payments to the publishers.

A syndicator or publisher’s server observes a “click” simply as a browser request for a URL associated with a particular ad. The server has no way to determine if a human initiated the action—and, if a human was involved, whether she acted knowingly and with honest intent. Syndicators typically seek to filter fraudulent or spurious clicks based on information such as the type of advertisement that was requested, the cost of the associated keyword, the IP address of the request and the recent number of requests from this address. In this paper, we propose an alternative approach. Rather than seeking to detect and eliminate fraudulent clicks, i.e., filtering out seemingly bad clicks, we consider ways of *authenticating valid clicks*, i.e., admitting only verifiably good ones. We refer to such validated clicks as *premium clicks*.

Our scheme involves a new entity, referred to as an *attestor*, that provides cryptographic credentials for clients that perform qualifying actions, such as purchases. These credentials allow the syndicator to distinguish premium clicks—corresponding to relatively low-risk clients—from other, general click traffic. Such classification of clicks strengthens a syndicator’s heuristic isolation of fraud risks.

The premium-click techniques that we describe in this paper are complementary to existing, filter-based tools for validating clicks: The two approaches can operate side by side.

**Organization.** We begin with a problem statement and a description of the related work in section 2, followed by a structural overview of our approach in section 3. In section 4, we outline our scheme and detail its technical foundations. We describe a prototype implementation of our scheme in section 5 and discuss user privacy in section 6, proposing several privacy-enhancing techniques. We provide a brief security analysis in section 7, and conclude in section 8. The paper appendix describes design choices for premium-click systems with multiple attestors.

## 2 Problem Overview and Related Work

*Click-fraud* is a type of abuse that exploits the lack of verifiable human engagement in PPC requests in order to fabricate ad traffic. It can take a number of forms. One virulent, automated type of click fraud involves a client that fraudulently simulates a click by means of a script or bot—or as the result of infection by a virus or Trojan. Such malware typically resides on the computer of the user from which the click will be generated, but can also in principle reside on access points and consumer routers [8, 9, 7]. Some click-fraud relies on *real* clicks, whether intentional or not. An example of the former is a so-called click-farm, which is a term denoting a group of low-wage workers who click for a living; another example involves deceiving or convincing users to click on advertisements. An example of an *unintentional* click is one generated by a malicious cursor-following script that places the banner right under the mouse cursor [6]. This can be done in a very small window to avoid detection. When the user clicks, the click would be interpreted as a click on the banner, and cause revenue generation to the attacker. A related abuse is manifested in an attack where publishers manipulate web pages such that honest visitors inadvertently trigger clicks [4]. This can be done for many common PPC schemes, and simply relies on the inclusion of a JavaScript component on the publisher’s webpage, where the script reads the banner and performs a get request that corresponds to what would be performed if a user had initiated a click.

Click fraud can benefit a fraudster in at least three known ways: First of all, a fraudster can use click-fraud to inflate the revenue of a publisher. Second, a fraudster can employ click-fraud to inflate advertising costs for a commercial competitor. As advertisers generally specify caps on their daily advertising expenses, such fraud is essentially a denial-of-service attack. Third, a fraudster can modify the ranking of advertisements by a combination of impressions and clicks. An impression is the viewing of the banner, with no click; this causes the ranking of the associated advertisement to go down. This can be done to benefit own advertising programs at the cost of those of competitors, and to manipulate the price paid per click for selected keywords.

Syndicators can in principle derive financial benefit from click fraud in the short term, as they receive revenue for whatever clicks they deem “valid.” In the long term, however, as customers become sensitive to losses, and syndicators rely on third-party auditors to lend credibility to their operations, click fraud can jeopardize syndicator-advertiser relationships. Thus syndicators ultimately have a strong incentive to eliminate fraudulent clicks. Today they employ a battery of filters to weed out suspicious clicks. These filters are trade secrets, as their dis-

closure might prompt new forms of fraud [10]. To give one example, though, it is likely that syndicators use IP tracing to determine if an implausible number of clicks is originating from a single source. While heuristic filters are fairly effective, they are of limited utility against sophisticated fraudsters, and subject to degraded performance as fraudsters learn to defeat them.

## 3 Structural Overview

**Authentication.** Our premium-click scheme is based on authentication of requests via cryptographic attestations on client behavior. We refer to these attestations as *coupons*. While a coupon could be realized straightforwardly using traditional *third-party cookies*, such cookies are so commonly blocked by consumers that their use is often impractical. Our scheme could alternatively involve traditional first-party cookies dispensed and harvested by a central authority. This architectural approach, however, presents limitations that we explain in depth in section 4.1. As we explain, we instead focus in this paper on the alternative mechanism of *cache cookies*.

Our premium-click scheme has two distinctive aspects:

1. **Pedigree:** Our scheme relies on designated Web sites called *attestors* to identify and label clients that appear to be operated by legitimate users—as opposed to bots or fraudsters. For example, an attestor might be a retail Web site that classifies as legitimate any client that has made at least \$50 of purchases. (Financial commitment here corroborates legitimate user behavior.) We refer to such clients, the producers of premium clicks in our scheme, as *premium clients*. In a loose sense, we propose the creation of an implicit reputation network to combat click-fraud, much like the seller reputation on eBay [3].
2. **Traffic caps:** Our scheme supports validation of clicks from clients that have not produced an excessive degree of click-traffic and thereby indicated possible malicious activity. In our approach, a click is only regarded as valid if accompanied by a coupon. Thus, we can detect multiple requests from the same origin by keeping track of coupon presentation. In the standard approach in which attestations like coupons do not play a role, detection of same-source traffic is more challenging, and often depends upon coarser origination data, such as IP addresses, or more fragile markers of continuity, such as session identifiers.

**Architecture.** In a traditional scheme, as a user clicks on a banner placed on the site of a publisher, the corresponding advertisement is downloaded from the advertiser and the transaction recorded by the syndicator. Later, the syndicator bills the advertiser and pays the publisher.

Under the model of premium clicks, there are additional tasks carried out: As a user performs a qualified action (such as a purchase), the corresponding attestation is embedded in his browser by an attestor. This attestation is released to the syndicator when the user clicks on a banner. The release can be initiated either by the syndicator or the advertiser. (Our prototype relies on syndicator triggering of coupon release.) The syndicator can pay attestors for their participation in a number of ways, ranging from a flat fee per time period to a payment that depends on the number of associated attestations that were recorded in a time interval. To avoid a situation where dishonest attestors issue larger number of attestations than the protocol prescribes (which would increase the earnings of the dishonest attestors), it is possible to appeal to standard auditing techniques.

**Challenges.** Our approach to premium clicks gives rise to two technical challenges. First, we must securely validate premium clients and their associated clicks. In other words, we must ensure that adversaries cannot impersonate premium clients or forge premium clicks. For this purpose, we apply basic cryptographic tools for data integrity. Second, we must protect the privacy of clients. While we do want syndicators to be able to authenticate clients, we do not want syndicators to be able to track them, learn their identities, or harvest side information about their browsing patterns. Toward this end, we propose ways in which coupons may be created as essentially anonymous credentials.

Of course, our techniques do not prevent misuse of coupons by clients that are “good,” i.e., controlled by honest users, and then turn “bad,” e.g., become infected with malware. By identifying the sources of clicks, however, and making traffic caps more effective, coupons in our scheme still offer some protection against fraud even in such cases.

It is important to observe that existing filtering methods cannot in general employ cookies/coupons to detect fraudulent clicks. That is because filtering is an *exclusionary* process: It seeks to identify and eliminate “bad” clicks. If a cookie were used to mark and exclude certain types of “bad” users, fraudsters could simply remove the cookies from their browsers. In contrast, because our premium-click scheme is *distinguishing*, i.e., it only accepts “good” clicks, it can benefit from the use of cookies/coupons. Cookies serve to mark “good” users.

## 4 A Premium-Click Scheme

In a world of perfect transparency, in which a syndicator knew the (real-world) identity of all users clicking on ads, click fraud would be much more manageable. In such a world, it would be easier to identify misbehavior by a real user—e.g., implausibly many clicks—as well as clicks initiated by bogus users or bots. A syndicator could go further, and reference databases containing profiles on the users who clicked on its published ads. The syndicator could even create a highly refined pricing structure based on a user’s predicted value as a potential consumer, with differential compensation for publishers. Our premium-click protocol diverges from this ideal in two senses:

- **Partial knowledge:** Given the fragmented nature of databases on user behavior and the privacy concerns attendant on user profiling, our overall profiling goal is modest. We would like to enable a syndicator only to determine that a click originates with a true human user with probable honest intent. We do not mainly focus on stronger differentiation among users, although our protocols could support this goal.
- **The browser as carrier:** Rather than relying on a central data repository, we rely on users’ browsers to convey information among participating sites. This approach helps eliminate engineering complexity and protect user privacy.

We design our premium-click scheme to support out-sourced PPC advertising. It can equally well secure against click fraud when ads are published directly on search engines: We need simply treat the syndicator and publisher as the same entity. The steps in our scheme are as follows and are illustrated in Figure 1. For simplicity, we assume a single syndicator  $\mathcal{S}$  and attestor  $\mathcal{A}$ . (We discuss the case of multiple attestors in the appendix.)

1. **Marking:** Based on its criteria for user validation, the attestor identifies a visiting client as legitimate. The attestor then “marks” the client. It does so by caching in the client’s browser a *coupon*  $\gamma$ , a kind of cryptographic token.
2. **Click / coupon release:** When a user clicks on a publisher’s advertisement in a browser, the user’s browser is directed to a URL on the syndicator’s site. This URL includes the publisher’s identity  $ID_{pub}$  and the identity of the advertisement that was clicked with  $ID_{ad}$ . The syndicator then causes the browser to release its coupon  $\gamma$  simultaneously with  $ID_{pub}$  and  $ID_{ad}$ .<sup>2</sup> We let  $C = (\gamma, ID_{pub}, ID_{ad})$

denote the released triple. We shall henceforth refer to  $\gamma$  or  $C$  alternately as a “coupon,” according to context.

3. **Coupon checking:** On receiving a triple  $C = (\gamma, ID_{pub}, ID_{ad})$ , the syndicator checks that  $\gamma$  is a (cryptographically) well formed coupon, as we describe in depth later. The syndicator also checks that the coupon has not been over-used, i.e., that  $C$  has not been submitted an excessive number of times in the recent past. (What constitutes “excessive” submission is a policy decision.)
4. **Reward:** If the syndicator successfully verifies that  $C$  represents a valid premium click, then the syndicator pays the publisher accordingly.

Of course, the publisher might embed additional information in  $C$ , e.g., a timestamp, etc. Moreover, a user’s browser might in fact contain multiple coupons  $\gamma_1, \gamma_2, \dots$  from different attestors, a possibility that we discuss below. A single computer may have multiple users, of course. If they each maintain a separate account, then their individual browser instantiations will carry user-specific coupons. When users share a browser, the browser may carry coupons if at least one of the users is validated by an attestor. While validation is not user-specific in this case, it is still helpful: A shared machine with a valid user is considerably more likely to see honest use than one without.

We now detail the technical foundations of our scheme. We assume here that the browser of a given user carries at most one coupon. We address the case of multiple coupons in the appendix.

## 4.1 Coupon caching

Our first technical design choice is the transport medium for coupons. To ensure its correct association with the browser that created it, a coupon is best communicated as a cached browser value (rather than through a back channel). At the same time, it is important to ensure that coupons be set such that only the syndicator can retrieve them, and fraudsters cannot easily harvest them.

Third-party cookies are the most obvious way to instantiate coupons. A third-party cookie is one set for a domain other than the one being visited by the user; thus, a coupon could be set as a third-party cookie. Because third-party cookies have a history of abusive application, however, users regularly block them. First-party cookies are an alternative mechanism. If an attestor redirects users to the site of a syndicator and provides user-specific or session-specific information in the redirection, then the syndicator can implant a coupon in the form of a

first-party cookie for its own, later use. Redirection of this kind, however, can be cumbersome, particularly if an attestor has relationships with multiple syndicators.

Cache cookies [5], particularly the TIF-based variety, offer an attractive alternative. An attestor can embed a coupon in a cache-cookie that is tagged for the site of a syndicator, i.e., exclusively readable by the syndicator. In their ability to be set for third-party sites, cache cookies are similar in functionality to third-party cookies. Cache cookies have a special, useful quirk, though: Any Web site visited by a user can cause them to be released to the site for which they are tagged. (Thus, as we shall see, it is important to authenticate the site initiating their release from a user’s browser.) Cache cookies, moreover, function even in browsers where ordinary cookies have been blocked. Cache cookies are therefore our preferred medium for coupons.

Briefly, a TIF-based cache cookie works as follows. Suppose we wish to set a cache cookie bearing value  $\gamma$  for release to Web site `www.S.com`. The cache cookie, then, assumes the form of an HTML page `ABC.html` that requests a resource from `www.S.com` bearing the value  $\gamma$ . For example, `ABC.html` might display a GIF image of the form `http://www.S.com/ $\gamma$ .gif`. Observe that *any* Web site can create `ABC.html` and plant it in a visiting user’s browser. Similarly *any* Web site that knows the name of the page/cache-cookie `ABC.html` can reference it, causing `www.S.com` to receive a request for  `$\gamma$ .gif`. Only `www.S.com`, however, can receive the cache cookie, i.e., the value  $\gamma$ , when it is released from the browser.

## 4.2 Coupon authentication

Ensuring against fraudulent creation or use of coupons is a key challenge in our scheme. Only attestors should be able to construct valid coupons. Coupons must therefore carry a form of cryptographic authentication. While digital signatures can in principle offer a flexible way to authenticate coupons, their computational costs are probably prohibitively expensive for a high-traffic, potentially multi-site scheme of the type we propose here. Message-authentication codes (MACs), a symmetric-key analog of digital signatures, are a more practical alternative.<sup>3</sup>

Suppose that the attestor  $\mathcal{A}$  and syndicator  $\mathcal{S}$  share a symmetric key  $k$ . (This key may be established out of band or using existing secure channels.) Let  $MAC_k(m)$  represent a strong message authentication code, e.g., HMAC [2], computed on a suitably formed message  $m$ . It is infeasible for any third party, e.g., an adversary, to generate a fresh MAC on any message  $m$ . Consequently, if a coupon assumes the form  $\gamma = m \parallel MAC_k(m)$  for a bitstring  $m$  that is unique to the visit of a client to the site of an attestor, then the coupon can be copied, but cannot

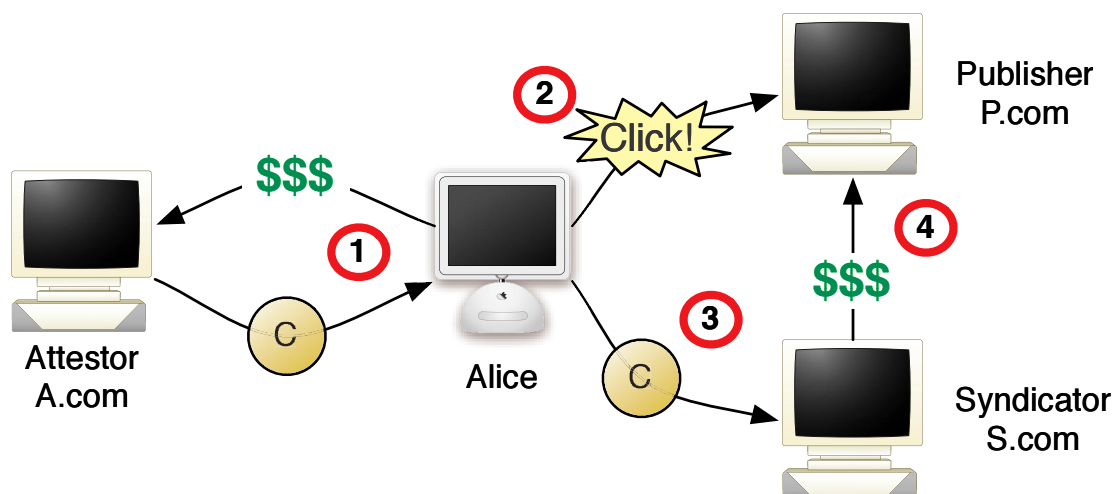


Figure 1: (1) **Alice** visits attestor **A.com**, spends money, and receives coupon value  $C = \gamma$ . (2) **Alice** visits publisher **P.com** and clicks on an ad. (3) **Alice's** browser transmits coupon  $C = (\gamma, ID_{pub}, ID_{ad})$  to syndicator **S.com**. (4) **S.com** pays **P.com** for Alice's premium click. (S.com also redirects Alice's browser to the entity that created the ad.)

be feasibly modified by a third party. The value  $m$  might be a suitably long (say, 128-bit) random nonce generated by  $\mathcal{A}$ . We propose some privacy-protecting alternative formats for  $m$  below.

### 4.3 Publisher identification/authentication

In addition to ensuring that a coupon is authentic, a syndicator must also be able to determine what publisher caused it to be released and is to receive payment for the associated click. Recall from above that a coupon takes the form  $C = (\gamma, ID_{pub}, ID_{ad})$ , where  $ID_{pub}$  is the identity of the publisher and  $ID_{ad}$  identifies the advertisement clicked. In order to create a full coupon, we must append  $ID_{pub}$  and  $ID_{ad}$  to  $\gamma$  as it is released. To do so, we can enhance a cache cookie webpage  $X.html$  to include the document *referrer*, i.e., the tag that identifies the webpage that causes its release. (In our scheme, this webpage is a URL on the syndicator,  $www.S.com$ , where both  $ID_{ad}$  and  $ID_{pub}$  are in the URL.) For example,  $X.html$  might take the following form:

```
<html><body>
<script language="JavaScript">
//Determine referring webpage r
// (which contains  $ID_{ad}$  and  $ID_{pub}$ ):
var r = escape(document.referrer);
//Write HTML to release the coupon  $\gamma.gif$ :
document.write('');
</script> </body> </html>$ 
```

Now when the syndicator's site page with a URL containing  $ID_{pub}$  and  $ID_{ad}$  references  $X.html$ , the syn-

dicator  $www.S.com$  receives a request for the resource  $\gamma.gif?ref=www.S.com\%3fad\%3d\langle ID_{ad} \rangle\%26pub\%3d\langle ID_{pub} \rangle$  (the value of the *ref* querystring variable in this resource request is the referrer, or page that triggered  $X.html$  to load, but encoded so it can appear in the URL). In essence, he receives a request for an image  $\gamma.gif$ , and is provided one querystring-style parameter containing the IDs of the advertisement and publisher. This string conveys the full desired coupon data  $C = (\gamma, ID_{pub}, ID_{ad})$ .

**Remark:** In cases where JavaScript is disabled by a client, an alternative approach is possible. An attestor can create not *one* cache cookie, but an array of cache cookies on independently created values  $\gamma_1^{(0)}, \dots, \gamma_k^{(0)}$  and  $\gamma_1^{(1)}, \dots, \gamma_k^{(1)}$ . To encode an  $k$ -bit publisher value  $ID_{pub} = b_1 \parallel \dots \parallel b_k$ , the publisher releases cache cookies corresponding to  $\gamma_1^{(b_1)}, \dots, \gamma_k^{(b_k)}$ . Of course, this method is somewhat more cumbersome than use of document-referrer strings, as it requires the syndicator to receive and correlate  $k$  distinct cache cookies for a single transaction.

### 4.4 Freshness

Authentication alone is insufficient to guarantee valid coupon use. It is also imperative to confirm that a coupon is *fresh*, that is, that a client is not replaying it more rapidly than justified by ordinary use.

To ensure coupon freshness, a syndicator may maintain a data structure  $T = \{R^{(1)}, \dots, R^{(r)}\}$  recording coupons received within a recent period of time (as deter-

mined by syndicator policy). A record  $R^{(i)}$  can include an authentication value  $\gamma^{(i)}$ , publisher identity  $ID_{pub}^{(i)}$ , ad identifier  $ID_{ad}^{(i)}$ , and a time of coupon receipt  $t^{(i)}$ .

When a new coupon  $C = (\gamma, ID_{pub}, ID_{ad})$  is received at time  $t$ , the syndicator can check whether there exists a  $C^{(i)} = (\gamma, ID_{pub}, ID_{ad}) \in T$  with time-stamp  $t^{(i)}$ . If  $t - t^{(i)} < \tau_{replay}$ , for some system parameter  $\tau_{replay}$  determined by syndicator policy, then the syndicator might reject  $C$  as a replay. Similarly, the syndicator can set replay windows for cross-domain and cross-advertisement clicks. For example, if  $C^{(i)} = (\gamma, ID_{pub}^{(i)}, ID_{ad}^{(i)})$ , where  $ID_{ad} \neq ID_{ad}^{(i)}$ , i.e., it appears that a given user has clicked on a different ad on the same site as that represented by  $C$ , the syndicator might implement a different check  $t - t^{(i)} < \tau_{crossclick}$  to determine that a coupon is stale and should be rejected. Since a second click on a given site is more likely representative of true user intent than a “doubleclick,” we would expect  $\tau_{crossclick} < \tau_{replay}$ .

Of course, many different filtering policies are possible, as are many different data structures and maintenance strategies for  $T$ .

## 5 Prototype Implementation

We implemented a prototype of our premium-click scheme. Four websites at separate IP addresses provide a simulated advertiser, publisher, attestor, and syndicator. The web sites are served by Apache 2.0.58, and server-side scripted with PHP 5.1.6. The database for click, ad, and coupon data is MySQL 5.0.26.

**Advertiser.** The prototype advertiser consists of two fabricated product pages designed as destinations for a user that clicks on a web ad. The only other duties of an Advertiser in the premium clicks system are to submit the ads to the syndicator, and then pay for billed clicks.

**Publisher.** The prototype publisher is a simple site that embeds ads, served by the syndicator, in iframes. Many widespread advertisement schemes (including Google’s AdSense) use this technique; others simply write directly to a publisher’s page, submitting their advertisements to the same origin as the publisher, thus making the scheme vulnerable to more click-fraud techniques [4].

**Attestor.** The prototype attestor is a simple service that provides a login box. When a user of the service provides a valid ID and password, he is provided an internal page that serves a cache cookie to the visitor’s browser. This simple HTML file is transmitted from the attestor to the visitor only once after login. Any subsequent requests

for the cache-cookie URL are replied to with an HTTP 304 “not modified” response. This forces the browser to use a cached version of the cookie if it exists, and does not provide one to browsers lacking a cached version of the cookie.

The cache cookie served by the attestor references an image hosted on the syndicator. The URL used to request the image is created by JavaScript when the cookie’s HTML is rendered, and contains the secret  $\gamma$  (which is generated when the cache cookie is set) as well as the referrer page, i.e., whichever page caused the cache cookie to load. Later, when the cookie is loaded in conjunction with a click, the URL of the referrer will reveal the ID of the publisher and the ID of the advertisement that was clicked.

The attestor needs to create and serve these cache cookies when the user logs in, so additional processing is required. However, creating a secret value takes very little time, and the cookie can be served in a hidden iframe. The result is no difference in experience for the user, and only a trivial amount of work for the attestor’s servers.

**Syndicator.** Of all the entities, the syndicator does the most work. It receives coupons released by the cache cookies (in the form of requested images), verifies the secrets in the coupons, and records clicks. Additionally, each ad click must be directed “through” the syndicator, so it must also serve a transfer page to direct the client’s web browser to the advertiser’s site. This is an ordinary flow of traffic in ad-serving systems that briefly delegates control to the syndicator who records clicks.

- *Database.* The syndicator hosts a MySQL database to house the advertisement data (content, advertiser’s ID, URL), click-through log (each click as it occurs, including advertisement ID and publisher ID), as well as a log of received coupons. Since the released coupon history needs to be saved and searched, we chose to use a database to ease development.
- *Processing Clicks.* When an advertisement is clicked, the client’s browser navigates to the syndicator’s site, bringing along the advertisement ID and the publisher ID. For example:  
`http://syndicator/click.php?ad=x&pub=y.`  
The syndicator then records the click, and responds with a web page that causes cache cookies from all attestors to load. (We only implemented one attestor, so one iframe is rendered with its content being the attestor’s cache cookie. When there are  $N$  possible attestors,  $N$  iframes are used.) Attestors’ cache cookies not available in the browser’s cache are simply not loaded. Coupons are released by the client’s browser from any attestor’s cache cookies.

- *Receiving Coupons.* Coupons are received by the syndicator in the form of requests for an image called “coupon.gif”. When this is requested, it is accompanied by a querystring. For example:

```
http://syndicator/coupon.gif?
secret= $\gamma$ &ref= $x$ .
```

The `ref` variable in the query string reflects both the publisher ID  $ID_{pub}$  and the advertisement  $ID_{ad}$  that was clicked. Receiving this request, the syndicator records the time, secret  $\gamma$  and referrer  $x$  in the database, and then serves a tiny image back to the client. HTTP headers are provided that force the client’s browser always to request this image, and not load it from cache. The purpose of this process is to ensure the coupons are always freshly delivered, and not loaded from browser cache.

- *Analyzing Clicks.* On the syndicator’s click-through page where the attestors’ cache-cookie iframes are present, a small delay is forced by JavaScript to allow the coupons transit time, since they load asynchronously in iframes. Immediately following that, the server decides if a click should be classified as “premium.” This is done by looking through the coupon database for recently released coupons corresponding to the advertisement that was clicked. Time between when the click was recorded and when the coupons arrived is noted, and only coupons within a pre-set window ( $\tau_{replay}$ , sixty seconds in our prototype) are considered in determining the premium status. If coupons are present and the secrets are valid, the click is recorded as “premium.”<sup>4</sup> Otherwise it is recorded as a general-class click.

In a production system, click analysis would be done after redirecting the client by adding it to a processing queue.

## 6 Privacy

In deploying our premium-click scheme with multiple attestors,  $\mathcal{A}_1, \dots, \mathcal{A}_q$ , it would be natural for a syndicator to share a unique key  $k_i$  with each attestor  $\mathcal{A}_i$ . Given such independent attestor keys  $\{k_i\}$ , though, a coupon created by  $\mathcal{A}_i$  conveys and therefore reveals the fact that a user has visited the Web site of  $\mathcal{A}_i$ . Observe, however, that in our scheme a publisher triggers the release of a coupon from the browser of a visiting user, but does not see the coupon. The syndicator receives the coupon, but does not directly interact with the user. In effect, the syndicator receives the coupon *blindly*. While the syndicator does learn the IP address of the user, this is information that is typically already available: The only additional information that the syndicator learns is whether

or not the user has received an attestation. Thus, coupons naturally decouple information about the browsing patterns of users from the identities and browsing sessions of users. This is an important, privacy-preserving feature.

Such decoupling occurs in the case when ads are outsourced, that is, when the syndicator and publisher are separate. When the syndicator and publisher are identical, i.e., when a search engine displays its own advertisements, coupons may be linked to users, and therefore leak potentially sensitive information. A couple of privacy-enhancing measures are possible. To limit the amount of leaked browsing implementation, our scheme may employ a multiple-coupon technique discussed in depth in Appendix A. Alternatively, attestors may share a single key  $k$  (or attestors may have overlapping sets of keys). In this case, a MAC does not reveal the identity of the attestor that created it. If a coupon  $\gamma = m \parallel MAC_k(m)$  is created, as we propose, with a random nonce  $m$ , then it conveys no information about a user’s identity. In principle, however, it would be possible for an attestor to embed a user’s identity in  $m$ , thereby transmitting it to the syndicator. This transmission could even be covert: A ciphertext on a user’s identity, i.e., an encryption thereof, will have the appearance of a random string. Proper auditing of the policy and operations of the attestor or syndicator would presumably be sufficient in most cases to ensure against collusive privacy infringements of this kind.

As an alternative,  $m$  might be based on distinctive, but verifiably non-identifying values. For example,  $m$  might include the IP address<sup>5</sup> and/or timestamp of the client to which an attestor issues a coupon—perhaps supplemented by a small counter value.<sup>6</sup> A client could then verify that  $m$  was properly formatted, and did not encode the user’s identity. Of course,  $MAC_k(m)$  itself might then embed the user’s identity. It is possible, however, to eliminate the possibility of a covert channel in the MAC by periodically refreshing  $k$  and publicly revealing old values.

### Remarks:

- There are good business motivations for attestors not merely to validate, but also to classify users. For example, an retailer might not merely indicate in a coupon that a client has spent enough to justify validation, but also provide a rough indication of much the client has spent (“low spender,” “medium spender,” “profligate”). Advertisers might then be charged on a differential basis according to the associated, perceived value of a client. Such classification would create a new dimension of privacy infringement. In the outsourcing case, where coupons

are decoupled from user identities, this approach might meet with user and regulator acceptance. In the case where the syndicator publishes advertisements, and coupons are linked to users, privacy is of greater concern. As advertisers will necessarily learn the syndicator's differential pricing scheme, there will be at least some transparency.

- In principle, public-key digital signatures offer more flexible privacy protection for coupon origins than MACs. For example, group signatures, e.g., [1], permit the identity of a signing attester to be hidden in the general case, but revoked by a trusted entity in the case of system compromise. In a high traffic advertising system, however, public-key cryptography would be prohibitively resource-intensive.

## 7 Security Analysis

Without possession of an attester key, an adversary cannot feasibly forge new coupons, thanks to our use of MACs. An adversary could still bypass our scheme in several ways:

- **Direct publisher fraud:** Using a slight modification of the proposed solution, the publisher could cause release of coupons even when users do not click on ads.
- **Indirect publisher fraud:** A dishonest Web site could re-direct users to the publisher's site.
- **Malware-driven clicks:** A virus or widely spread Trojan could either surreptitiously direct a user's browser to a Web site and simulate a click or else steal a coupon from the browser for use on another platform.

All of these attacks are possible in existing click-fraud schemes. The various techniques used to address them today are equally applicable to premium clicks. For example, a syndicator can direct its own client machines to a publisher's site to determine if the publisher is generating fraudulent clicks. Indeed, our premium-click scheme makes detection of misbehavior easier, as it permits a syndicator to "mark" a client coupon and therefore directly monitor the traffic generated by the client and even detect the emergence of stolen coupons.

An adversary can also try to exploit the special characteristics of our scheme as follows:

- **Posing as an attester:** An adversary might either establish itself as an attester or compromise the key of an existing attester. If the syndicator sets appropriate policies for creating attestors, then it should

be difficult for an adversary to pose as one. Attestors are likely, in any case, to be a more exclusive class of Web site than publishers or even advertisers. Moreover, in the case where MAC keys are attester-specific, the syndicator can individually monitor the traffic generated by each attester, making fraud detection easier.

- **Compromise of an attester key:** An adversary can attempt to learn the MAC key of an existing attester. The difficulty of this form of attack depends on the security of the attester's Web site. MAC keys for premium clicks may be protected using many of the same measures employed to secure SSL keys and other cryptographic secrets.
- **Coupon harvesting:** An adversary could harvest coupons from attestors by creating accounts or clients that meet their validation criteria. By establishing appropriate policies for validation by its attestors, the syndicator can attempt to attach a financial cost to this form of fraud in excess of the gains that a fraudster might reap from it.

## Auditing

Since the syndicator is ultimately in control over deciding which clicks should be considered "premium" (and earns more when clicks *are* premium), publishers and advertisers may accuse the syndicator of improperly inflating the percentage of clicks considered premium. To solve this problem, an additional entity called an *auditor* can be contracted to watch the coupons that are released, and verify the premium-status judgement of the syndicator. The auditor would not be rewarded based on click traffic, so it would have no incentive to inflate or deflate the number of premium clicks from those that are legitimate.

The cache cookies set by attestors can be crafted so that, when an advertisement's URL is clicked, the coupon  $C = (\gamma, ID_{pub}, ID_{ad})$  is released both to the syndicator *and* to the auditor who maintains an independent database. When the syndicator's numbers are contested, the coupons recorded by the auditor can be used to recompute the number of premium clicks for a given advertisement or publisher, and compared to the syndicator's calculation.

## 8 Conclusion

In contrast to today's heuristic filtering methods for eliminating "bad" clicks, our premium-click scheme relies on a foundation of cryptographic authentication to validate "good" clicks. Premium clicks are by no means a cure-all for fraud, and are themselves subject to attack. The

value of premium clicks lies in the way that they provide new, cryptographically authenticated visibility into click traffic, and thus a new, stronger platform for combating click fraud.

While premium clicks could in principle supplant current filtering schemes entirely, they are attractive in that they can be deployed in a complementary fashion alongside existing systems. We have proposed a new advertising model in which advertisers pay a higher charge for premium clicks. We believe that such a scheme might be launched experimentally by a syndicator with minimal impact on existing business and then expanded as its success warrants. Thus premium clicks promise offer not only a new approach to click fraud, but one with a practical path to fruition.

## References

- [1] ATENIESE, G., CAMENISCH, J., JOYE, M., AND TSUDIK, G. A practical and provably secure coalition-resistant group signature scheme. In *Advances in Cryptology-Crypto '00* (2000), M. Bellare, Ed., Springer. LNCS vol. 1880.
- [2] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Keying hash functions for message authentication. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, 1996), Springer-Verlag, pp. 1–15.
- [3] EBAY. [www.ebay.com](http://www.ebay.com), Accessed 31 January 2007.
- [4] GANDHI, M., JAKOBSSON, M., AND RATKIEWICZ, J. Badvertisements: Stealthy click-fraud with unwitting accessories. In *Anti-Phishing and Online Fraud, Part I Journal of Digital Forensic Practice, Volume 1, Special Issue 2* (November 2006).
- [5] JUELS, A., JAKOBSSON, M., AND JAGATIC, T. Cache cookies for browser authentication (extended abstract). In *IEEE Symposium on Privacy and Security* (2006), pp. 301–305.
- [6] RSNAKE. Stealing mouse clicks for banner fraud. <http://ha.ckers.org/blog/20070116/stealing-mouse-clicks-for-banner-fraud/>, January 2007.
- [7] STAMM, S., RAMZAN, Z., AND JAKOBSSON, M. drive-by pharming, 2006. Technical Report, <http://www.cs.indiana.edu/pub/techreports/TR641.pdf>.
- [8] TSOW, A. Phishing with consumer electronics – malicious home routers. In *In Models of Trust for the Web, a workshop at the 15th International World Wide Web Conference (WWW2006)* (2006).
- [9] TSOW, A., JAKOBSSON, M., YANG, L., AND WETZEL, S. Warkitting: the drive-by subversion of wireless home routers. In *Anti-Phishing and Online Fraud, Part II Journal of Digital Forensic Practice, Volume 1, Special Issue 3* (November 2006).
- [10] TUZHILIN, A. The Lanes gifts v. Google report, 2006. Independent evaluators assessment of quality of Googles click-fraud filtering methods. Accessed 31 January 2007 at [http://googleblog.blogspot.com/pdf/Tuzhilin\\_Report.pdf](http://googleblog.blogspot.com/pdf/Tuzhilin_Report.pdf).

## Notes

<sup>1</sup>This research was performed by the author at RavenWhite Inc.

<sup>2</sup>The reason for having the syndicator trigger coupon-release is twofold: (1) To prevent JavaScript-based click automation, ads today are often rendered inside an iframe whose source is loaded from the syndicator's site and (2) To eliminate the need for JavaScript on the publisher's site.

<sup>3</sup>An authenticator could create a list  $X$  of random codes and transfer it to the syndicator via a backchannel, but this would not be efficient (and would also eliminate some of the privacy properties we would like to achieve).

<sup>4</sup>In a multi-attestor environment, where clients may carry and release multiple coupons, the syndicator needs some mechanism to determine which coupons correspond to a given client. A simple option is to attach a fresh, random number (nonce) to the links in each rendered advertisement. The nonce will attach itself to all of the coupons that a client releases in a given click.

<sup>5</sup>Inclusion of an IP address in a coupon also has some security benefits. In the case where the syndicator publishes its own ads, it can check that a client's presented IP address is consistent with the IP address in the coupon, e.g., it originates with the same service provider.

<sup>6</sup>A counter might still embed a covert channel, but, if the size of the channel might be made small enough to alleviate the problem of privacy infringement significantly.

## A Multiple Attestors

User privacy in our premium-click scheme depends upon how the value  $\gamma$  is formed, and on the number and content of the coupons cached in a user's browser. Let us now therefore consider a system with multiple attestors,  $A_1, \dots, A_q$ . Each attestor  $auth_i$  shares a key  $k_i$  with the syndicator. We now describe the technical challenges that arise with multiple attestors.

**Multiple coupons.** The first problem we encounter in a system with multiple attestors is the difficulty of managing multiple cache cookies across different domains. A cache-cookie system can involve caching of a set of  $j$  different webpages  $X_1, X_2, \dots, X_j$  in a given user's browser, each webpage serving as a *slot* for a distinct cache cookie. Two difficulties arise, however. The first is that a site seeking to release a set of cache cookies (i.e., the publisher) cannot determine what slots in a user's browser actually contain cache cookies. The only way for the publisher to release all cache cookies is to call all  $j$  webpages. The second is that a site seeking to set a cache cookie, i.e., an attestor, cannot determine if a given slot has been filled. If the attestor plants a cache cookie in a slot that already contains one, the previously planted cache cookie will be effaced.

The simplest way to circumvent these difficulties in our premium-clicks scheme is to manage only a single slot, that is, to maintain only a single cache cookie in a given user's browser. Only the cache cookie planted most recently by an attestor will then persist. Provided that the syndicator regards all attestors as having equal authority

in validating users, this approach does not result in any service degradation.

If, however, the syndicator desires the ability to harvest multiple coupons, then attestors must use multiple slots. One possible approach is to maintain an individual slot for each attestor, i.e., to let  $j = q$ . If the number of attestors is small, this may be workable. Alternatively, attestors may plant coupons in random slots, sometimes supplanting previous coupons, or subsets of attestors may share slots. The syndicator might, for example, assign different weight to attestors, according to the anticipated reliability of their attestations; attestors with the same rating might share a slot.

**Keying.** One approach to management of attestor keys is to assign an identical key  $k$  to all attestors, i.e., let  $k_1 = k_2 \dots = k$ . While this approach has the merit of simplicity, it has the disadvantage of rendering tracing and key-revocation difficult.

It is preferable, therefore to create attestor keys  $\{k_i\}$  in an independent manner. In this case, a coupon  $\gamma = m \parallel MAC_{k_i}(m)$  is cryptographically bound to the attestor that created it. That is, only attestor  $\mathcal{A}_i$ , with its knowledge of  $k_i$ , can feasibly create  $\gamma$  of this form. To enable the syndicator to determine the correct key for verification of the MAC, the coupon must be supplemented with  $i$ , the identity of the authenticator. For example, we might let  $m = i \parallel r$ , where  $r$  is a random nonce.

# SpyProxy: Execution-based Detection of Malicious Web Content

Alexander Moshchuk, Tanya Bragin, Damien Deville,  
Steven D. Gribble, and Henry M. Levy  
*Department of Computer Science & Engineering*  
*University of Washington*  
{anm, tbragin, damien, gribble, levy}@cs.washington.edu

## Abstract

This paper explores the use of **execution-based** Web content analysis to protect users from Internet-borne malware. Many anti-malware tools use signatures to identify malware infections on a user's PC. In contrast, our approach is to render and observe active Web content in a disposable virtual machine **before** it reaches the user's browser, identifying and blocking pages whose behavior is suspicious. Execution-based analysis can defend against undiscovered threats and zero-day attacks. However, our approach faces challenges, such as achieving good interactive performance, and limitations, such as defending against malicious Web content that contains non-determinism.

To evaluate the potential for our execution-based technique, we designed, implemented, and measured a new proxy-based anti-malware tool called SpyProxy. SpyProxy intercepts and evaluates Web content in transit from Web servers to the browser. We present the architecture and design of our SpyProxy prototype, focusing in particular on the optimizations we developed to make on-the-fly execution-based analysis practical. We demonstrate that with careful attention to design, an execution-based proxy such as ours can be effective at detecting and blocking many of today's attacks while adding only small amounts of latency to the browsing experience. Our evaluation shows that SpyProxy detected every malware threat to which it was exposed, while adding only 600 milliseconds of latency to the start of page rendering for typical content.

## 1 Introduction

Web content is undergoing a significant transformation. Early Web pages contained simple, passive content, while modern Web pages are increasingly *active*, containing embedded code such as ActiveX components, JavaScript, or Flash that executes in the user's browser. Active content enables a new class of highly interactive applications, such as integrated satellite photo/mapping

systems. Unfortunately, it also leads to new security threats, such as "drive-by-downloads" that exploit browser flaws to install malware on the user's PC.

This paper explores a new *execution-based* approach to combating Web-borne malware. In this approach we render and execute Web content in a disposable, isolated execution environment before it reaches the user's browser. By observing the side-effects of the execution, we can detect malicious behavior in advance in a safe environment. This technique has significant advantages: because it is based on behavior rather than signatures, it can detect threats that have not been seen previously (e.g., zero-day attacks). However, it raises several crucial questions as well. First, can execution-based analysis successfully detect today's malware threats? Second, can the analysis be performed without harming browser responsiveness? Third, what are the limitations of this approach, in particular in the face of complex, adversarial scripts that contain non-determinism?

Our goal is to demonstrate the potential for execution-based tools that protect users from malicious content as they browse the Web. To do this, we designed, prototyped, and evaluated a new anti-malware service called SpyProxy. SpyProxy is implemented as an extended Web proxy: it intercepts users' Web requests, downloads content on their behalf, and evaluates its safety before returning it to the users. If the content is unsafe, the proxy blocks it, shielding users from the threat. Our intention is not to replace other anti-malware tools, but to add a new weapon to the user's arsenal; SpyProxy is complementary to existing anti-malware solutions.

SpyProxy combines two key techniques. First, it executes Web content on-the-fly in a disposable virtual machine, identifying and blocking malware before it reaches the user's browser. In contrast, many existing tools attempt to remove malware after it is already installed. Second, it monitors the executing Web content by looking for suspicious "trigger" events (such as registry writes or process creation) that indicate potentially malicious activity [28]. Our analysis is therefore based on behavior rather than signatures.

SpyProxy can in principle function either as a service deployed in the network infrastructure or as a client-side protection tool. While each has its merits, we focus in this paper on the network service, because it is more challenging to construct efficiently. In particular, we describe a set of performance optimizations that are necessary to meet our goals.

In experiments with clients fetching malicious Web content, SpyProxy detected every threat, some of which were missed by other anti-spyware systems. Our evaluation shows that with careful implementation, the performance impact of an execution-based malware detector can be reduced to the point where it has negligible effect on a user's browsing experience. Despite the use of a "heavyweight" Internet proxy and virtual machine techniques for content checking, we introduce an average delay of only 600 milliseconds to the start of rendering in the client browser. This is small considering the amount of work performed and relative to the many seconds required to fully render a page.

The remainder of the paper proceeds as follows. Section 2 presents the architecture and implementation of SpyProxy, our prototype proxy-based malware defense system. Section 3 describes the performance optimizations that we used to achieve acceptable latency. In section 4 we evaluate the effectiveness and performance of our SpyProxy prototype. Section 5 discusses related work and we conclude in Section 6.

## 2 Architecture and Implementation

This section describes SpyProxy—an execution-based proxy system that protects clients from malicious, active Web objects. We begin our discussion by placing SpyProxy in the context of existing malware defenses and outlining a set of design goals. We next describe the architecture of SpyProxy and the main challenges and limitations of our approach.

### 2.1 Defending Against Modern Web Threats

Over the past several years, attackers have routinely exploited vulnerabilities in today's Web browsers to infect users with malicious code such as spyware. Our crawler-based study of Web content in October 2005 found that a surprisingly large fraction of pages contained drive-by-download attacks [28]. A drive-by-download attack installs spyware when a user simply visits a malicious Web page.

Many defenses have been built to address this problem, but none are perfect. For example, many users install commercial anti-spyware or anti-virus tools, which are typically signature-based. Many of these tools look only for malware that is already installed, attempting the difficult operation of removing it after the fact. Firewall-based network detectors can filter out some well-known

and popular attacks, but they typically rely on static scanning to detect exploits, limiting their effectiveness. They also require deployment of hardware devices at organizational boundaries, excluding the majority of household users. Alternatively, users can examine blacklists or public warning services such as SiteAdvisor [41] or Stop-Badware [43] before visiting a Web site, but this can be less reliable [5, 44].

None of these defenses can stop zero-day attacks based on previously unseen threats. Furthermore, signature databases struggle to keep up with the rising number of malware variants [9]. As a result, many of today's signature-based tools fail to protect users adequately from malicious code on the Web.

### 2.2 Design Goals

SpyProxy is a new defense tool that is designed for today's Web threats. It strives to keep Web browsing convenient while providing on-the-fly protection from malicious Web content, including zero-day attacks. Our SpyProxy architecture has three high-level goals:

1. **Safety.** SpyProxy should protect clients from harm by preventing malicious content from reaching client browsers.
2. **Responsiveness.** The use of SpyProxy should not impair the interactive feel and responsiveness of the user's browsing experience.
3. **Transparency.** The existence and operation of SpyProxy should be relatively invisible and compatible with existing content-delivery infrastructure (both browsers and servers).

Providing safety while maintaining responsiveness is challenging. To achieve both, SpyProxy uses several content analysis techniques and performance-enhancing optimizations that we next describe.

### 2.3 Proxy-based Architecture

Figure 1 shows the architecture of a simplified version of SpyProxy. Key components include the *client browser*, *SpyProxy*, and remote *Web servers*. When the client browser issues a new request to a Web server, the request first flows through SpyProxy where it is checked for safety.

When a user requests a Web page, the browser software generates an HTTP request that SpyProxy must intercept. Proxies typically use one of two methods for this: browser configuration (specifying an HTTP proxy) or network-level forwarding that transparently redirects HTTP requests to a proxy. Our prototype system currently relies on manual browser configuration.

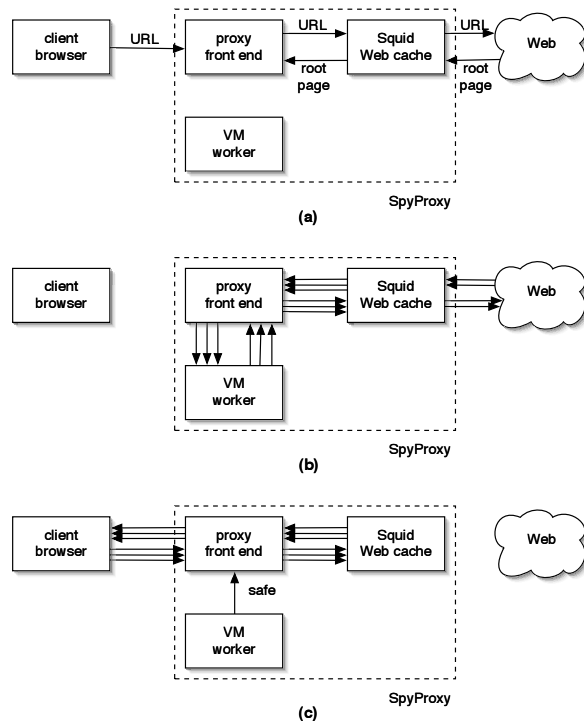


Figure 1: **SpyProxy architecture.** (a) A client browser requests a Web page; the proxy front end intercepts the request, retrieves the root page, and statically analyzes it for safety. (b) If the root page cannot be declared safe statically, the front end forwards the URL to a VM worker. A browser in the VM downloads and renders the page content. All HTTP transfers flow through the proxy front end and a Squid cache. (c) If the page is safe, the VM notifies the front end, and the page content is released to the client browser from the Squid cache. Note that if the page has been cached and was previously determined to be safe, the front end forwards it directly to the client.

The SpyProxy front end module receives clients' HTTP requests and coordinates their processing, as shown in Figure 1(a). First, it fetches the root page using a cache module (we use Squid in our prototype). If the cache misses, it fetches the data from the Web, caching it if possible and then returning it to the front end. Second, the front end statically analyzes the page (described below) to determine whether it is safe. If safe, the proxy front end releases the root page content to the client browser, and the client downloads and renders it and any associated embedded objects.

If the page cannot be declared safe statically, the front end sends the page's URL to a virtual machine (VM) worker for dynamic analysis (Figure 1(b)). The worker directs a browser running in its VM to fetch the requested URL, ultimately causing it to generate a set of HTTP requests for the root page and any embedded objects. We configure the VM's browser to route these requests

first through the front end and then through the locally running Squid Web cache. Routing it through the front end facilitates optimizations that we will describe in Section 3. Routing the request through Squid lets us reduce interactions with the remote Web server.

The browser in the VM worker retrieves and renders the full Web page, including the root page and all embedded content. Once the full page has been rendered, the VM worker informs the front end as to whether it has detected suspicious activity; this is done by observing the behavior of the page during rendering, as described below. If so, the front end notifies the browser that the page is unsafe. If not, the front end releases the main Web page to the client browser, which subsequently fetches and downloads any embedded objects (Figure 1(c)).

### 2.3.1 Static Analysis of Web Content

On receiving content from the Internet, the SpyProxy front end first performs a rudimentary form of static analysis, as previously noted. The goal of static analysis is simple: if we can verify that a page is safe, we can pass it directly to the client without a sophisticated and costly VM-based check. If static analysis were our only checking technique, our analysis tool would need to be complex and complete. However, static analysis is just a performance optimization. Content that can be analyzed and determined to be safe is passed directly to the client; content that cannot be passed to a VM worker for additional processing.

Our static analyzer is conservative. If it cannot identify or process an object, it declares it to be potentially unsafe and submits it to a VM worker for examination. For example, our analyzer currently handles normal and chunked content encodings, but not compressed content. Future improvements to the analyzer could reduce the number of pages forwarded to the VM worker and therefore increase performance.

When the analyzer examines a Web page, it tries to determine whether the page is *active* or *passive*. Active pages include executable content, such as ActiveX, JavaScript, and other code; passive pages contain no such interpreted or executable code. Pages that contain active content must be analyzed dynamically.

It is possible for seemingly passive content to compromise the user's system if the renderer has security holes. Such flaws have occurred in the past in both the JPEG and PNG image libraries. For this reason, we consider any non-HTML content types to be unsafe and send them for dynamic processing. In principle, a browser's HTML processor could have vulnerabilities in it as well; it is possible to configure SpyProxy to disable all static checking if this is a concern.

We validated the potential benefits of static checking with a small measurement study, where we collected a

17-hour trace of Web requests generated by the user population in our department. We saw that 54.8% of HTML pages transferred contain passive content. Thus, there can be significant benefit in identifying these pages and avoiding our VM-based check for them.

### 2.3.2 Execution-based Analysis through VM-based Page Rendering

A drive-by download attack occurs when a Web page exploits a flaw in the victim's browser. In the worst case, an attack permits the attacker to install and run arbitrary software on the victim's computer. Our execution-based approach to detecting such attacks is adapted from a technique we developed in our earlier spyware measurement study [28], where we used virtual machines to determine whether a Web page had malicious content. We summarize this technique here.

Our detection method relies on the assumption that malicious Web content will attempt to break out of the security sandbox implemented by the browser. For example, the simple act of rendering a Web page should never cause any of the following side-effects: the creation of a new process other than known helper applications, modifications to the file system outside of safe folders such as the browser cache, registry modifications, browser or OS crashes, and so on. If we can determine that a Web page triggers any of these unacceptable conditions, we have proof that the Web page contains malicious content.

To analyze a Web page, we use a "clean" VMware [45] virtual machine configured with unnecessary services disabled. We direct an unmodified browser running in the VM to fetch and render the Web page. Because we disabled other services, any side effects we observe must be caused by the browser rendering the Web page. We monitor the guest OS and browser through "triggers" installed to look for sandbox violations, including those listed above. If a trigger fires, we declare the Web page to be unsafe. This mechanism is described in more detail in [28].

Note that this technique is behavior-based rather than signature-based. We do not attempt to characterize vulnerabilities; instead, we execute or render content to look for evidence of malicious side-effects. Accordingly, given a sufficiently comprehensive set of trigger conditions, we can detect zero-day attacks that exploit vulnerabilities that have not yet been identified.

## 2.4 Limitations

Our approach is effective, but has a number of challenges and limitations. First, the overhead of cloning a VM, rendering content within it, and detecting trigger conditions is potentially high. In Section 3 we describe several optimizations to eliminate or mask this overhead,

and we evaluate the success of these optimizations in Section 4. Second, our trigger monitoring system should be located outside the VM rather than inside it, to prevent it from being tampered with or disabled by the malware it is attempting to detect. Though we have not done so, we believe we could modify our implementation to use techniques such as VM introspection [18] to accomplish this. Third, pre-executing Web content on-the-fly raises several correctness and completeness issues, which we discuss below.

### 2.4.1 Non-determinism

With SpyProxy in place, Web content is rendered twice, once in the VM's sandboxed environment and once on the client. For our technique to work, all attacks must be observed by the VM: the client must never observe an attack that the VM-based execution missed. This will be true if the Web content is deterministic and follows the same execution path in both environments. In this way, SpyProxy is ideal for deterministic Web pages that are designed to be downloaded and displayed to the user as information.

However, highly interactive Web pages resemble general-purpose programs whose execution paths depend on non-deterministic factors such as randomness, time, unique system properties, or user input. An attacker could use non-determinism to evade detection. For example, a malicious script could flip a coin to decide whether to carry out an attack; this simple scheme would defeat SpyProxy 50% of the time.

As a more pertinent example, if a Web site relies on JavaScript to control ad banner rotation, it is possible that the VM worker will see a benign ad while the client will see a malicious ad. Note, however, that much of Internet advertising today is served from ad networks such as DoubleClick or Advertising.com. In these systems, a Web page makes an image request to the server, and any non-determinism in picking an ad happens on the server side. In this case, SpyProxy will return the same ad to both the VM worker and the client. In general, only client-side non-determinism could cause problems for SpyProxy.

There are some potential solutions for handling non-determinism in SpyProxy. Similar to ReVirt [12], we could log non-deterministic events in the VM and replay them on the client; this likely would require extensive browser modifications. We could rewrite the page to make it deterministic, although a precise method for doing this is an open problem, and is unlikely to generalize across content types. The results of VM-based rendering can be shipped directly to the client using a remote display protocol, avoiding client-side rendering altogether, but this would break the integration between the user's browser and the rest of their computing environment.

None of these approaches seem simple or satisfactory; as a result, we consider malicious non-determinism to be a fundamental limitation to our approach. In our prototype, we did not attempt to solve the non-determinism problem, but rather we evaluated its practical impact on SpyProxy's effectiveness. Our results in Section 4 demonstrate that our system detected all malicious Web pages that it examined, despite the fact that the majority of them contained non-determinism. We recognize that in the future, however, an adversary could introduce non-determinism in an attempt to evade detection by SpyProxy.

#### 2.4.2 Termination

Our technique requires that the Web page rendering process terminates so that we can decide whether to block content or forward it to the user. SpyProxy uses browser interfaces to determine when a Web page has been fully rendered. Unfortunately, for some scripts termination depends on timer mechanisms or user input, and in general, determining when or whether a program will terminate is not possible.

To prevent "timebomb-based" attacks, we speed up the virtual time in the VM [28]. If the rendering times out, SpyProxy pessimistically assumes the page has caused the browser to hang and considers it unsafe. Post-rendering events, such as those that fire because of user input, are not currently handled by SpyProxy, but could be supported with additional implementation. For example, we could keep the VM worker active after rendering and intercept the events triggered because of user input to forward them to the VM for pre-checking. The interposition could be accomplished by inserting run-time checks similar to BrowserShield [33].

#### 2.4.3 Differences Between the Proxy and Client

In theory, the execution environment in the VM and on the client should be identical, so that Web page rendering follows the same execution path and produces the same side-effects in both executions. Differing environments might lead to false positives or false negatives.

In practice, malware usually targets a broad audience and small differences between the two environments are not likely to matter. For our system, it is sufficient that harmful side-effects produced at the client are a subset of harmful side-effects produced in the VM. This implies that the VM system can be partially patched, which makes it applicable for all clients with a higher patch level. Currently, SpyProxy uses unpatched Windows XP VMs with an unpatched IE browser. As a result, SpyProxy is conservative and will block a threat even if the client is patched to defend against it.

There is a possibility that a patch could contain a bug, causing a patched client to be vulnerable to an attack to

which the unpatched SpyProxy is immune [24]. We assume this is a rare occurrence, and do not attempt to defend against it.

### 2.5 Client-side vs. Network Deployment

As we hinted before, SpyProxy has a flexible implementation: it can be deployed in the network infrastructure, or it can serve as a client-side proxy. There are many tradeoffs involved in picking one or the other. For example, a network deployment lets clients benefit from the workloads of other clients through caching of both data and analysis results. On the other hand, a client-side approach would remove the bottleneck of a centralized service and the latency of an extra network hop. However, clients would be responsible for running virtualization software that is necessary to support SpyProxy's VM workers. Many challenges, such as latency optimizations or non-determinism issues, apply in both scenarios.

While designing our prototype and carrying out our evaluation, we decided to focus on the network-based SpyProxy. In terms of effectiveness, the two approaches are identical, but obtaining good performance with a network deployment presents more challenges.

## 3 Performance Optimizations

The simple proxy architecture described in section 2 will detect and block malicious Web content effectively, but it will perform poorly. For a given Web page request, the client browser will not receive or render any content until the proxy has downloaded the full page from the remote Web server, rendered it in a VM worker, and satisfied itself that no triggers have fired. Accordingly, many of the optimizations that Web browsers perform to minimize perceived latency, such as pipelining the transfer of embedded objects and the rendering of elements within the main Web page, cannot occur.

To mitigate the cost of VM-based checking in our proxy, we implemented a set of performance optimizations that either enable the browser to perform its normal optimizations or eliminate proxy overhead altogether.

### 3.1 Caching the Result of Page Checks

Web page popularity is known to follow a Zipf distribution [6]. Thus, a significant fraction of requests generated by a user population are repeated requests for the same Web pages. Web proxy caches take advantage of this fact to reduce Web traffic and improve response times [1, 13, 15, 21, 52, 53]. Web caching studies generally report hit rates as high as 50%.

Given this, our first optimization is caching the *result* of our security check so that repeated visits to the same page incur the overhead of our VM-based approach only

once. In principle, the hit rate in our security check cache should be similar to that of Web caches.

This basic idea faces complications. The principle of complete mediation warns against caching security checks, since changes to the underlying security policy or resources could lead to caching an incorrect outcome [34]. In our case, if any component in a Web page is dynamically generated, then different clients may be exposed to different content. However, in our architecture, our use of the Squid proxy ensures that no confusion can occur: we cache the result of a security check only for objects that Squid also caches, and we invalidate pages from the security cache if any of the page's objects is invalid in the Squid cache. Thus, we generate a hit in the security cache only if all of the Web page content will be served out of the Squid proxy cache. Caching checks for non-deterministic pages is dangerous, and we take the simple step of disabling the security cache for such pages.

### 3.2 Prefetching Content to the Client

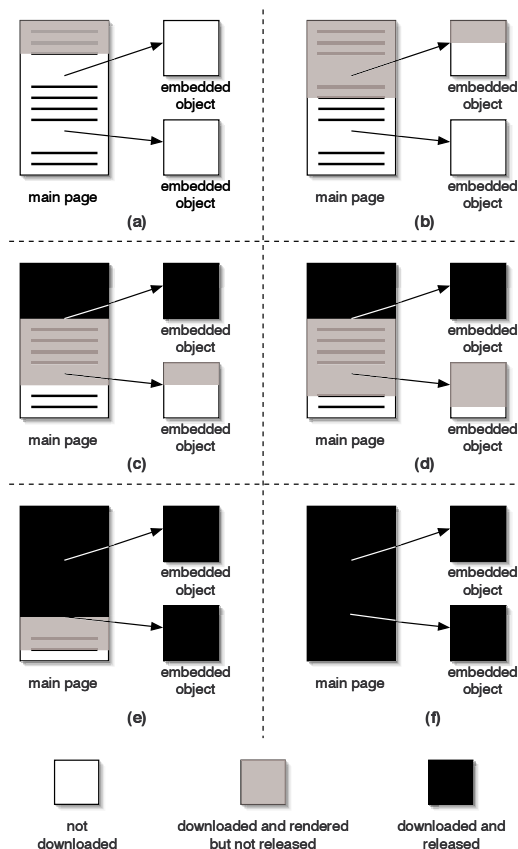
In the unoptimized system shown in Figure 1, the Web client will not receive any content until the entire Web page has been downloaded, rendered, and checked by SpyProxy. As a result, the network between the client and the proxy remains idle when the page is being checked. If the client has a low-bandwidth network connection, or if the Web page contains large objects, this idle time represents a wasted opportunity to begin the long process of downloading content to the client.

To rectify this, SpyProxy contains additional components and protocols that overlap several of the steps shown in Figure 1. In particular, a new client-side component acts as a SpyProxy agent. The client-side agent both prefetches content from SpyProxy and releases it to the client browser once SpyProxy informs it that the Web page is safe. This improves performance by transmitting content to the client in parallel with checking the Web page in SpyProxy. Because we do not give any Web page content to the browser before the full page has been checked, this optimization does not erode security.

In our prototype, we implemented the client-side agent as an IE plugin. The plugin communicates with the SpyProxy front end, spooling Web page content and storing it until SpyProxy grants it authorization to release the content to the browser.

### 3.3 The Staged Release of Content

Although prefetching allows content to be spooled to the client while SpyProxy is performing its security check, the user's browser cannot begin rendering any of that content until the full Web page has been rendered and checked in the VM worker. This degrades responsiveness, since the client browser cannot take advantage



**Figure 2: Staged release optimization.** The progression of events in the VM worker's browser shows how staged release operates on a Web page with two embedded objects. As embedded objects become fully downloaded and rendered by the VM worker's browser, more of the Web page is released to the client-side browser.

of its performance optimizations that render content well before the full page has arrived.

We therefore implemented a "staged release" optimization. The goal of staged release is to present content considered safe for rendering to the client browser in pieces; as soon as the proxy believes that a slice of content (e.g., an object or portion of an HTML page) is safe, it simultaneously releases and begins transmitting that content to the client.

Figure 2 depicts the process of staged release. A page consists of a root page (typically containing HTML) and a set of embedded objects referred to from within the root page. As a Web browser downloads and renders more and more of the root page, it learns about embedded objects and begins downloading and rendering them.

Without staged release, the proxy releases no content until the full Web page and its embedded objects have been rendered in the VM. With staged release, once the VM has rendered an embedded object, it releases that

object and all of the root page content that *precedes* its reference. If the client browser evaluates the root page in the same order as the VM browser, this is safe to do; our results in Section 4 confirm this optimization is safe in practice. Thus, a pipeline is established in which content is transmitted and released incrementally to the client browser.

In Figures 2(a) and 2(b), only part of the main Web page has been downloaded and rendered by the VM browser. In Figure 2(c), all of the first embedded object has been rendered by the VM, which causes that object and some of the main Web page content (shown in black) to be released and transmitted to the client browser. More of the main Web page and the second embedded object is downloaded and rendered in Figure 2(d), until finally, in Figures 2(e) and 2(f), the full Web page is released.

Many Web pages contain dozens of embedded images. For example, CNN’s Web page contains over 32 embedded objects. Faced with such a Web page, our staged release optimization quickly starts feeding the client browser more and more of the root page and associated embedded objects. As a result, the user does not observe expensive Web access delay.

Note that staged release is independent from prefetching. With prefetching, content is pushed to the client-side agent before SpyProxy releases it to the client browser; however, no content is released until the full page is checked. With staged release, content is released incrementally, but released content is not prefetched. Staged release can be combined with prefetching, but since it does not require a client-side agent to function, it may be advantageous to implement staged release without prefetching. We evaluate each of these optimizations independently and in combination in Section 4.

### 3.4 Additional optimizations

SpyProxy contains a few additional optimizations. First, the VM worker is configured to have a browser process already running inside, ready to accept a URL to retrieve. This avoids any start-up time associated with booting the guest OS or launching the browser. Second, the virtual disk backing the VM worker is stored in a RAM-disk file system in the host OS, eliminating the disk traffic associated with storing cookies or files in the VM browser. Finally, instead of cloning a new VM worker for every client request, we re-use VM workers across requests, garbage collecting them only after a trigger fires or a configurable number of requests has occurred. Currently, we garbage collect a worker after 50 requests.

## 4 Evaluation

This section evaluates the effectiveness and performance of our SpyProxy architecture and prototype. The

malicious pages visited	browser exploits	27
	spontaneous downloads	73
	total	100
# sites containing the malicious pages		45
<b>malicious pages blocked by SpyProxy</b>		<b>100%</b>
malicious domains identified by SiteAdvisor		80%
malicious pages containing non-determinism		96%

Table 1: **Effectiveness of SpyProxy.** The effectiveness of SpyProxy at detecting and blocking malicious Web content. SpyProxy was successful at detecting and blocking 100% of the malicious Web pages we visited, in spite of the fact that most of them contained non-determinism. In comparison, the SiteAdvisor service incorrectly classified 20% of the malicious Web domains as benign.

prototype includes the performance optimizations we described previously. Our results address three key questions: how effective is our system at detecting and blocking malicious Web content, how well do our performance optimizations mask latency from the user, and how well does our system perform given a realistic workload?

### 4.1 Effectiveness at Blocking Malicious Code

We first consider the ability of SpyProxy to successfully block malicious content. To quantify this, we manually gathered a list of 100 malicious Web pages on 45 distinct sites. Each of these pages performs an attack of some kind. We found these pages using a combination of techniques, including: (1) searching Google for popular Web categories such as music or games, (2) mining public blacklists of known attack sites, and (3) examining public warning services such as SiteAdvisor.

Some of the Web pages we found exploit browser vulnerabilities to install spyware. Others try to “push” malicious software at clients spontaneously, requiring user consent to install it; we have configured SpyProxy to automatically accept such prompts to evaluate its effectiveness at blocking these threats. The pages include a diversity of attack methods, such as the WMF exploit, ActiveX controls, applet-based attacks, JavaScript, and pop-up windows. A successful attack inundates the victim with adware, dialer, and Trojan downloader software.

Table 1 quantifies the effectiveness of our system. SpyProxy detected and blocked 100% of the attack pages, despite the diversity of attack methods to which it was exposed. Further, most of these attack pages contained some form of non-deterministic content; in practice, none of the attacks we found attempted to evade detection by “hiding” inside non-deterministic code.

The table also shows the advantage of our on-the-fly approach compared to a system like SiteAdvisor, which provides static recommendations based on historical evidence. SiteAdvisor misclassified 20% of the malicious sites as benign. While we cannot explain why SiteAdvi-

sor failed on these sites, we suspect it is due to a combination of incomplete Web coverage (i.e., not having examined some pages) and stale information (i.e., a page that was benign when examined has since become malicious). SpyProxy’s on-the-fly approach examines Web page content as it flows towards the user, resulting in a more complete and effective defense.

For an interesting example of how SpyProxy works in practice, consider *www.crackz.ws*, one of our 100 malicious pages. This page contains a specially crafted image that exploits a vulnerability in the Windows graphics rendering engine. The exploit runs code that silently downloads and installs a variety of malware, including several Trojan downloaders. Many signature-based anti-malware tools would not prevent this attack from succeeding; they would instead attempt to remove the malware after the exploit installs it.

In contrast, when SpyProxy renders a page from *www.crackz.ws* in a VM, it detects the exploit when the page starts performing unacceptable activity. In this case, as the image is rendered in the browser, SpyProxy detects an unauthorized creation of ten helper processes. SpyProxy subsequently blocks the page before the client renders it. Note that SpyProxy does not need to know any details of the exploit to stop it. Equally important, in spite of the fact that the exploit attacks a non-browser flaw that is buried deep in the software stack, SpyProxy’s behavior-based detection allowed it to discover and prevent the attack.

## 4.2 Performance of the Unoptimized System

This section measures the performance of the basic *unoptimized* SpyProxy architecture we described in Section 2.3. These measurements highlight the limitations of the basic approach; namely, unoptimized SpyProxy interferes with the normal browser rendering pipeline by delaying transmission until an entire page is rendered and checked. They also suggest opportunities for optimization and provide a baseline for evaluating the effectiveness of those optimizations.

We ran a series of controlled measurements, testing SpyProxy under twelve configurations that varied across the following three dimensions:

- **Proxy configuration.** We compared a regular browser configured to communicate directly with Web servers with a browser that routes its requests through the SpyProxy checker.
- **Client-side network.** We compared a browser running behind an emulated broadband connection with a browser running on the same gigabit Ethernet LAN as SpyProxy. We used the client-side NetLimiter tool and capped the upload and download client

	Google		NY Times		MSN blog	
	render begins	render ends	render begins	render ends	render begins	render ends
direct	0.21s	0.64s	0.41s	4.8s	0.40s	10.2s
unoptimized SpyProxy	0.79s	1.2s	3.4s	7.3s	2.7s	12.4s

(a) broadband

	Google		NY Times		MSN blog	
	render begins	render ends	render begins	render ends	render begins	render ends
direct	0.20s	0.63s	0.41s	3.3s	0.36s	2.3s
unoptimized SpyProxy	0.79s	1.2s	3.4s	5.3s	2.7s	3.9s

(b) gigabit

**Table 2: Performance of the unoptimized SpyProxy.** These tables compare the latency of an unprotected browser that downloads content directly from Web servers to that of a protected browser downloading through the SpyProxy service. We show the latency until the page begins to render on the client and the latency until the page finishes rendering. The data are shown for three Web pages as well the client on (a) an emulated broadband access link, and (b) the same LAN as SpyProxy.

bandwidth at 1.5 Mb/s to emulate the broadband connection.

- **Web page requested.** We measured three different Web pages: the Google home page, the front page of the *New York Times*, and the “MSN shopping insider” blog, which contains several large, embedded images. The Google page is small: just 3,166 bytes of HTML and a single 8,558 byte embedded GIF. The *New York Times* front page is larger and more complex: 92KB of HTML, 74 embedded images, 4 stylesheets, 3 XML objects, 1 flash animation, and 10 embedded JavaScript objects. This represents 844KB of data. The MSN blog consists of a 79KB root HTML page, 18 embedded images (the largest of which is 176KB), 2 stylesheets, and 1 embedded JavaScript object, for a total of 1.4MB of data.

For each of the twelve configurations, we created a timeline showing the latency of each step from the client’s Web page request to the final page rendering in the client. We broke the end-to-end latency into several components, including WAN transfer delays, the overhead of rendering content in the VM worker before releasing it to the client, and internal communication overhead in the SpyProxy system itself. We cleared all caches in the system to ensure that content was retrieved from the original Web servers in all cases.

For each configuration, Table 2 shows the time until content first begins to render on the user’s screen and the time until the Web page finishes rendering. In all cases,

time (ms)	event
0	user requests URL, browser generates HTTP request
169	SpyProxy FE receives request, requests root page from Squid
538	SpyProxy FE finishes static check, forwards URL to VM
560	VM browser generates HTTP request
561	first byte of root page arrives at VM browser
3055	last byte of last page component arrives at VM browser
3363	VM browser finishes rendering, checking triggers
3374	first byte of root page arrives at client browser
7334	last byte of last page component arrives at client browser
7347	client browser finishes rendering content

client browser transfer and render time: 4.5s  
overhead introduced by VM browser: 2.8s  
other SpyProxy system overhead: 0.05s

**Table 3: Detailed breakdown of the unoptimized SpyProxy.** Events occurring when fetching the New York Times page over broadband through SpyProxy. Most SpyProxy overhead is due to serializing the VM browser download and trigger checks before transferring or releasing content to the client browser.

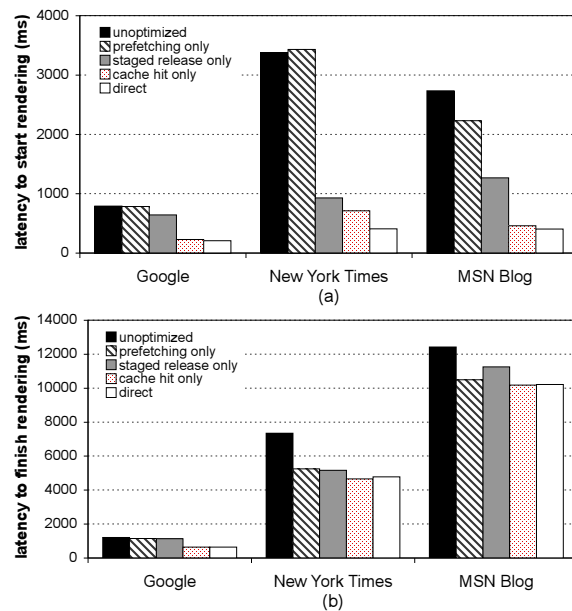
the unoptimized SpyProxy implementation added less than three seconds to the total page download time. However, the time until rendering began was much higher on the unoptimized system, growing in some cases by a factor of ten. This confirms that our system can perform well, but, without optimizations, it interferes with the browser’s ability to reduce perceived latency by pipelining the transfer and rendering of content.

Table 3 provides a more detailed timeline of events when fetching the *New York Times* page from a broadband client using the unoptimized SpyProxy. Downloading and rendering the page in the VM browser introduced 2.8 seconds of overhead. Since no data flows to the client browser until SpyProxy finishes rendering and checking content, this VM rendering latency is responsible for delay experienced by the user.

### 4.3 Performance Optimizations

To reduce the overhead introduced by the unoptimized SpyProxy system, we previously described three optimization techniques: prefetching content to a client-side agent, the staged release of content to the client browser, and caching the results of security checks. We now present the results of a set of microbenchmarks that evaluate the impact of each optimization.

Figure 3 summarizes the benchmark results. Both figures show the latency to download three different pages to a client on the emulated broadband connection. For each page, we show latency for five cases: (1) the unoptimized SpyProxy, (2) SpyProxy with only prefetching enabled, (3) SpyProxy with only staged release enabled, (4) SpyProxy with a hit in the enabled security cache, and (5) the base case of a client fetching content directly from Web servers. Figure 3(a) shows the latency before page rendering begins in the client browser, while Figure 3(b)



**Figure 3: Performance of optimizations (broadband).** The latency until the client browser (a) begins rendering the page, and (b) finishes rendering the page. Each graph shows the latency for three different pages for five configurations.

	Google		NY Times		MSN blog	
	render begins	render ends	render begins	render ends	render begins	render ends
unoptimized SpyProxy	0.79s	1.21s	3.37s	7.3s	2.7s	12.4s
prefetching only	.78s (-0.01s)	1.15s (-0.06s)	3.43s (+0.06s)	5.2s (-2.1s)	2.2s (-0.5s)	11.3s (-1.1s)

**Table 4: Prefetching (broadband).** Latency improvements gained by the prefetching optimization in the broadband environment. Prefetching alone did not yield significant benefits.

shows the latency until page rendering ends.

In combination, the optimizations serve to reduce the latency before the *start* of rendering in the client. With all of the the optimizations in place, the page load “feels” nearly as responsive through SpyProxy as it does without SpyProxy. In either case, the page begins rendering about a second after the request is generated. The optimizations did somewhat improve the total rendering latency relative to the unoptimized implementation (Figure 3(b)), but this was not nearly as dramatic. Page completion time is dominated by transfer time over the broadband network, and our optimizations do nothing to reduce this.

	Google		NY Times		MSN blog	
	render begins	render ends	render begins	render ends	render begins	render ends
unoptimized SpyProxy	0.79s	1.21s	3.37s	7.3s	2.7s	12.4s
staged release only	0.64s (-0.15s)	1.13s (-0.08s)	0.92s (-2.45s)	5.2s (-2.1s)	1.3s (-1.4s)	11.3s (-1.1s)

Table 5: **Staged release (broadband)**. Latency improvements from staged release in the broadband environment. Staged release significantly improved the latency until rendering starts. It yielded improvements similar to prefetching in the latency until full page rendering ends.

### 4.3.1 Prefetching

Prefetching by itself does not yield significant benefits. As shown in Table 4, it did not reduce rendering start-time latency. With prefetching alone, the client browser effectively stalls while the VM browser downloads and renders the page fully in the proxy. That is, SpyProxy does not release content to the client’s browser until the VM-based check ends.

However, we did observe some improvement in finish-time measurements. For example, the time to fully render the *New York Times* page dropped by 2.1 seconds, from 7.3 seconds in the unoptimized SpyProxy to 5.2 seconds with prefetching enabled. Prefetching successfully overlaps some transmission of content to the client-side agent with SpyProxy’s security check, slightly lowering overall page load time.

### 4.3.2 Staged Release

Staged release very successfully reduced initial latency before rendering started; this time period has the largest impact on perceived responsiveness. As shown in Table 5, staged release reduced this latency by several seconds for both the *New York Times* and MSN blog pages. In fact, from the perspective of a user, the *New York Times* page began rendering nearly four times more quickly with staged release enabled. For all three pages, initial rendering latency was near the one-second mark, implying good responsiveness.

The staged release optimization also reduced the latency of rendering the full Web page to nearly the same point as prefetching. Even though content does not start flowing to the client until it is released, this optimization releases some content quickly, causing an overlap of transmission with checking that is similar to prefetching.

Staged release outperforms prefetching in the case that matters—initial time to rendering. It also has the advantage of not requiring a client-side agent. Once SpyProxy decides to release content, it can simply begin uploading it directly to the client browser. Prefetching requires the installation of a client-side software compo-

	Google		NY Times		MSN blog	
	render begins	render ends	render begins	render ends	render begins	render ends
unoptimized SpyProxy	0.79s	1.21s	3.37s	7.3s	2.7s	12.4s
security cache hit	0.23s (-0.56s)	0.64s (-0.57s)	0.71s (-2.66s)	4.6s (-2.7s)	0.5s (-2.2s)	10.2s (-2.2s)

Table 6: **Security cache hit (broadband)**. This table shows the latency improvements gained when the security cache optimization is enabled and the Web page hits in the cache.

nent, and it provides benefits above staged release only in a narrow set of circumstances (namely, pages that contain very large embedded objects).

To better visualize the impact of staged release, Figure 4 depicts the sequence of Web object completion events that occur during the download and rendering of a page. Figure 4(a) shows completion events for the *New York Times* page. The unoptimized SpyProxy (top) does not transmit or release events to the client browser until the full page has rendered in the VM. With staged release (4(a) bottom), as objects are rendered and checked by the SpyProxy VM, they are released and transmitted to the client browser and then rendered. Accordingly, the sequence of completion events is pipelined between the two browsers. This leads to much more responsive rendering and an overall lower page load time.

Figure 4(b) shows a similar set of events for the MSN blog page. Since this page consists of few large embedded images, the dominant cost in both the unoptimized and staged-release-enabled SpyProxy implementations is the time to transmit the images to the client over broadband. Accordingly, though staged release permits the client browser to begin rendering more quickly, most objects queue up for transmission over the broadband link after being released by SpyProxy.

### 4.3.3 Caching

When a client retrieves a Web page using the optimized SpyProxy, both the outcome of the security check and the page’s content are cached in the proxy. When a subsequent request arrives for the same page, if any of its components are cached and still valid, our system avoids communicating with the origin Web server. In addition, if all components of the page are cached and still valid, the system uses the previous security check results instead of incurring the cost of a VM-based evaluation.

In Table 6, we show the latency improvement of hitting in the security cache compared with the unoptimized SpyProxy. As with the other optimizations, the primary benefit of the security cache is to improve the latency until the page begins rendering. Though the full page load time improves slightly, the transfer time over the broad-

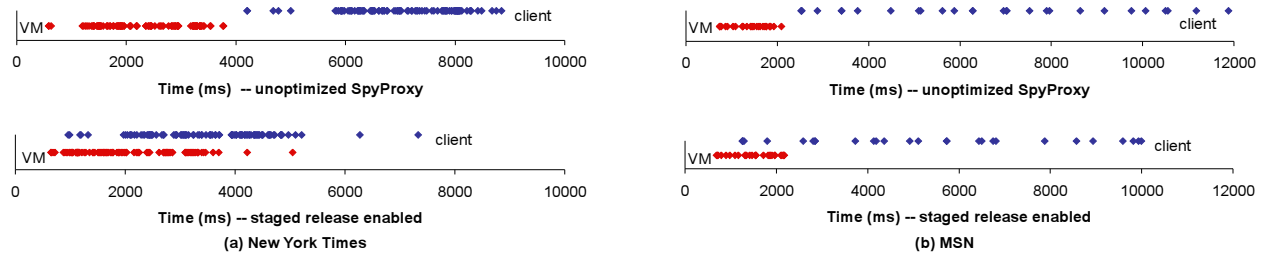


Figure 4: **Timeline of events with staged release (broadband).** The sequence of object rendering completion events that occur over time for (a) the New York Times, and (b) MSN blog pages. The top figures show the sequence of events for the unoptimized SpyProxy, while the bottom figures show what happens with staged release. In each figure, the top series of dots represents completions in the client browser and the bottom series in SpyProxy’s VM browser. Staged release is effective at the early release of objects to the browser.

band connection still dominates. However, on a security cache hit, the caching optimization is extremely effective, since it eliminates the need to evaluate content in a VM.

#### 4.4 Performance on a Realistic Workload

Previous sections examined the individual impact of each of our optimizations. In the end, however, the question remains: how does SpyProxy perform for a “typical” user Web-browsing workload? A more realistic workload will cause the performance optimizations — caching, static analysis, and staged release — to be exercised together in response to a stream of requests.

To study the behavior of SpyProxy when confronted with a realistic request stream, we measured the response latencies of 1,909 Web page requests issued by our broadband Web client. These requests were generated with a Zipf popularity distribution drawn from a list of 703 different safe URLs from 124 different sites. We chose the URLs by selecting a range of popular and unpopular sites ranked by the Alexa ranking service. By selecting real sites, we exercised our system with the different varieties and complexities of Web page content to which users are typically exposed. By generating our workload with a Zipf popularity distribution, we gave our caching optimization the opportunity to work in a realistic scenario. None of the sites we visited contained attacks; our goal was simply to evaluate the performance impact of SpyProxy on browsing.

Figure 5(a) presents a cumulative distribution function for the time to start page rendering in the client browser. This is the delay the user sees before the browser responds to a request. Figure 5(b) shows the CDF for full-page-load latencies. Each figure depicts distributions for three cases: (1) directly connecting to the Web site without SpyProxy, (2) using the optimized SpyProxy implementation, and (3) using SpyProxy with optimizations disabled. We flushed all caches before gathering the data for each distribution.

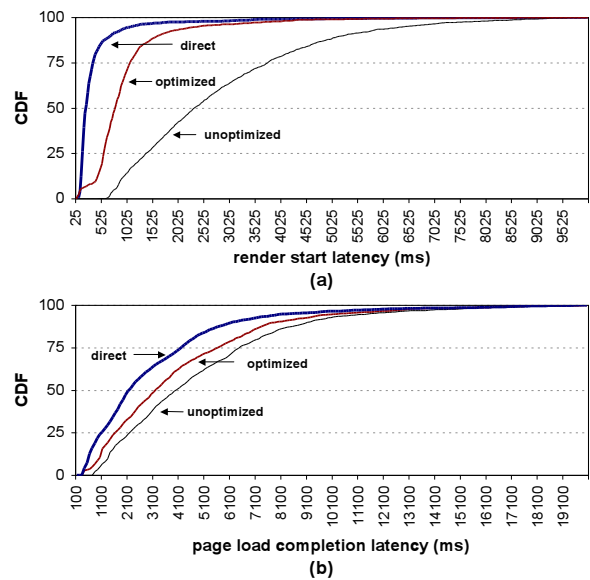


Figure 5: **Overall performance (broadband).** These graphs show the distributions of (a) render start latencies and (b) full page load latencies for a workload consisting of 1,909 requests issued to 703 pages from 102 Web sites. Each graph compares the response time for a direct client, the unoptimized system, and the fully optimized system. The artifact visible at low latencies on the optimized line in (a) corresponds to hits in our security cache.

Our results demonstrate that the optimized SpyProxy system delivers content to browsers very quickly. The median time until rendering began was 0.8 seconds in the optimized system compared to 2.4 seconds in the unoptimized system. There is still room to improve; the median start time for the direct connection was 0.2 seconds. However, the optimized system feels acceptably fast to a user. In contrast, the unoptimized system seems noticeably sluggish compared to the optimized system and direct connections.

A typical request flowing through the optimized sys-

tem involves several potential sources of overhead, including interacting with the Squid proxy cache and pre-executing content in a virtual machine. In spite of this, the optimized SpyProxy effectively masks latency, resulting in an interactive, responsive system. In addition, our system generated very few false positives: only 4 of the 1,909 Web page requests resulted in an alarm being raised. Even though the offending pages were benign, they did in fact attempt to install software on the user's computer, albeit by requesting permission from the user first. For example, one of the pages prompted the user to install a browser plug-in for the QuickTime media player. We chose not to deal with such opt-in installers, as SpyProxy is primarily intended for zero-day attacks that never ask for permission before installing malware. However, we do reduce the number of false positives by including the most common browser plug-ins, such as Flash, in the base VM image.

## 4.5 Scalability

SpyProxy is designed to service many concurrent users in an organizational setting. Our implementation runs on a cluster of workstations, achieving incremental scalability by executing VM workers on additional nodes. We now provide some back-of-the-envelope estimations of SpyProxy's scalability. We have not performed an explicit scaling benchmark, but our calculations do provide an approximate indication of how many CPUs would be necessary to support user population of a given size.

Our estimate is based on the assumption that the CPU is likely to be the bottleneck of a deployed system; for this to be true, the system must be configured with an adequate amount of memory and network bandwidth to support the required concurrent virtual machines and Web traffic. While performing the evaluation in section 4.4, we measured the amount of CPU time required to process a Web page in SpyProxy. On a 2.8GHz Pentium 4 machine with 4GB of RAM and a single 80GB 7200 RPM disk, we found the average CPU time consumed per page was 0.35 seconds.

There is little published data on the number of Web pages users view per day. In a study of Internet content-delivery systems [38], users requested 930 HTTP objects per day on average, and another study found that an average Web page contains about 15 objects [27]. Combining these, we conservatively estimate that a typical user browses through 100 pages per day. Assuming this browsing activity is uniformly distributed over an 8-hour workday, one CPU can process 82286 Web pages per day, implying a single-CPU SpyProxy could support approximately 822 users. A single quad-core machine should be able to handle the load from an organization containing a few thousand people.

## 4.6 Summary

This section evaluated the effectiveness and performance of our SpyProxy prototype. Our measurements demonstrated that SpyProxy effectively detects malicious content. In our experiments, SpyProxy correctly detected and blocked every threat, including several that SiteAdvisor failed to identify. Our experiments with fully optimized SpyProxy show that a proxy-based spyware checker can be implemented with only minimal performance impact on the user. On average, the use of SpyProxy added only 600 milliseconds to the user-visible latency before rendering starts. In our experience using the system, this small additional overhead does not noticeably degrade the system's responsiveness.

## 5 Related Work

We now discuss related research on spyware detection and prevention, intrusion detection and firewall systems, and network proxies.

### 5.1 Spyware and Malware Detection

In previous work, we used passive network monitoring to measure adware propagation on the University of Washington campus [37]. In a follow-on study, we used Web crawling to find and analyze executable programs and Web pages that lead to spyware infections [28]; the trigger-based VM analysis technique in that work forms the foundation for SpyProxy's detection mechanism.

Strider HoneyMonkey [49] and the commercial SiteAdvisor service [41] both use a VM-based technique similar to ours to characterize malicious Web sites and pages. Our work differs in two main ways: we show that our VM-based technique can be used to build a transparent defense system rather than a measurement tool, and we examine optimizations that enable our system to perform efficiently and in real time.

Our system detects malicious Web content by executing it and looking for evidence of malicious side-effects. Other systems have attempted to detect malware by examining side-effects, including Gatekeeper [51], which monitors Windows extensibility hooks for evidence of spyware installation. Another recent detector identifies spyware by monitoring API calls invoked when sensitive information is stolen and transmitted [20]. However, these systems only look for malware that is already installed. In contrast, SpyProxy uses behavioral analysis to *prevent* malware installation.

Other works have looked at addressing limitations of signature-based detection. Semantics-aware malware detection [8] uses an instruction-level analysis of programs to match their behavior against signature templates. This technique improves malware detection, but

not prevention. Several projects explore automatic generation of signatures for detection of unknown malware variants [7, 29, 40, 46, 48]. These typically need attack traffic and time to generate signatures, leaving some clients vulnerable when a new threat first appears.

Some commercial client-side security tools have begun to incorporate behavioral techniques, and two recent products, Prevx1 [32] and Primary Response Safe-Connect [35], use purely behavioral detection. However, these tools must run on systems packed with client-installed programs, which limits their behavioral analysis. In contrast, SpyProxy pre-executes content in a clean sandbox, where it can apply a much stricter set of behavioral rules.

Other approaches prevent Web-based malware infestations by protecting the user's system from the Web browser using VM isolation [10], OS-level sandboxing [16, 36], or logging/rollback [17]. Fundamentally, this *containment* approach is orthogonal to our *prevention* approach. Although these tools provide strong isolation, they have different challenges, such as data sharing and client-side performance overhead.

Remote playgrounds move some of the browser functionality (namely, execution of untrusted Java applets) away from the client desktop and onto dedicated machines [26]; the client browser becomes an I/O terminal to the actual browser running elsewhere. Our architecture is different — SpyProxy *pre-executes* Web pages using an unmodified browser and handles any form of active code, allowing it to capture a wider range of attacks. Nevertheless, SpyProxy could benefit from this technique in the future, for example by forwarding user input to the VM worker in AJAX sites.

Several projects tackle the detection and prevention of other classes of malware, including worms, viruses, and rootkits [19, 39, 50]. SpyProxy complements these defenses with protection against Web-borne attacks, resulting in better overall desktop security.

## 5.2 Intrusion Detection and Firewalls

Intrusion detection systems (e.g., Bro [31] and snort [42]) protect networks from attack by searching through incoming packets for known attack signatures. These systems are typically passive, monitoring traffic as it flows into a network and alerting a system administrator when an attack is suspected. More sophisticated intrusion detection systems attempt to identify suspicious traffic using anomaly detection [3, 4, 22, 23]. A related approach uses protocol-level analysis to look for attacks that exploit specific vulnerabilities, such as Shield [47]. The same idea has been applied at the HTML level in client-side firewalls and proxies [25, 30, 33].

These systems typically look for attack signatures for well-established protocols and services. As a result, they

cannot detect new or otherwise undiscovered attacks. Since they are traditionally run in a passive manner, attacks are detected but not prevented. Our system executes potentially malicious content in a sandboxed environment, using observed side-effects rather than signatures to detect attacks and protect clients.

Shadow honeypots combine network intrusion detection systems and honeypots [2]. They route risky network traffic to a heavily instrumented version of a vulnerable application, which detects certain types of attacks at run-time. In contrast, SpyProxy does not need to instrument the Web browser that it guards, and its run-time checks are more general and easier to define.

## 5.3 Proxies

Proxies have been used to introduce new services between Web clients and servers. For example, they have been used to provide scalable distillation services for mobile clients [14], Web caching [1, 13, 15, 21, 52, 53], and gateway services for onion-routing anonymizers [11]. SpyProxy builds on these advantages, combining active content checking with standard proxy caching. Spy-Bye [30] is a Web proxy that uses a combination of blacklisting, whitelisting, ClamAV-based virus scanning, and heuristics to identify potentially malicious Web content. In contrast, SpyProxy uses execution-based analysis to identify malicious content.

## 6 Conclusions

This paper described the design, implementation, and evaluation of SpyProxy, an execution-based malware detection system that protects clients from malicious Web pages, such as drive-by-download attacks. SpyProxy executes active Web content in a safe virtual machine *before* it reaches the browser. Because SpyProxy relies on the behavior of active content, it can block zero-day attacks and previously unseen threats. For performance, SpyProxy benefits from a set of optimizations, including the staged release of content and caching the results of security checks.

Our evaluation of SpyProxy demonstrates that it meets its goals of safety, responsiveness, and transparency:

1. SpyProxy successfully detected and blocked all of the threats it faced, including threats not identified by other detectors.
2. The SpyProxy prototype adds only 600 milliseconds of latency to the start of page rendering—an amount that is negligible in the context of browsing over a broadband connection.
3. Our prototype integrates easily into the network and its existence is transparent to users.

Execution-based analysis does have limitations. We described several of these, including issues related to non-determinism, termination, and differences in the execution environment between the client and the proxy.

There are many existing malware detection tools, and although none of them are perfect, together they contribute to a “defense in depth” security strategy. Our goal is neither to build a perfect tool nor to replace existing tools, but to add a new weapon to the Internet security arsenal. Overall, our prototype and experiments demonstrate the feasibility and value of on-the-fly, execution-based defenses against malicious Web-page content.

## References

- [1] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the IEEE Conference on Parallel and Distributed Information Systems (PDIS '96)*, Miami Beach, FL, December 1996.
- [2] Kostas G. Anagnostakis, Stelios Sidiroglou, Periklis Akritidis, Konstantinos Xinidis, Evangelos Markatos, and Angelos D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.
- [3] Kevin Borders and Atul Prakash. Web tap: Detecting covert Web traffic. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*, New York, NY, October 2004.
- [4] Kevin Borders, Xin Zhao, and Atul Prakash. Siren: Catching evasive malware (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Washington, DC, May 2006.
- [5] Tanya Bragin. Measurement study of the Web through a spam lens. Technical Report TR-2007-02-01, University of Washington, Computer Science and Engineering, February 2007.
- [6] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of 18th Annual IEEE Conference on Computer Communications (IEEE INFOCOM '99)*, March 1999.
- [7] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Washington, DC, May 2006.
- [8] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.
- [9] Andrew Conry-Murray. Product focus: Behavior-blocking stops unknown malicious code. [http://mirage.cs.ucr.edu/mobilecode/resources\\_files/behavior.pdf](http://mirage.cs.ucr.edu/mobilecode/resources_files/behavior.pdf), June 2002.
- [10] Richard Cox, Steven Gribble, Henry Levy, and Jacob Hansen. A safety-oriented platform for Web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Washington, DC, May 2006.
- [11] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.
- [12] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 2002.
- [13] Brian Duska, David Marwood, and Michael J. Feeley. The measured access characteristics of World Wide Web client proxy caches. In *Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems (USITS '97)*, Monterey, CA, December 1997.
- [14] Armando Fox, Steven Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, St.-Malo, France, October 1997.
- [15] Steven Glassman. A caching relay for the World Wide Web. *Computer Networks and ISDN Systems*, 27(2):165–173, 1994.
- [16] Green Border Technologies. GreenBorder desktop DMZ solutions. <http://www.greenborder.com>, November 2005.
- [17] Francis Hsu, Hao Chen, Thomas Ristenpart, Jason Li, and Zhendong Su. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*, Washington, DC, December 2006.
- [18] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.
- [19] Darrell Kienzle and Matthew Elder. Recent worms: A survey and trends. In *Proceedings of the 2003 ACM Workshop on Rapid Malcode (WORM '03)*, Washington, DC, October 2003.
- [20] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.
- [21] Tom M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the bounds of Web latency reduction from caching and prefetching. In *Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems (USITS '97)*, Monterey, CA, December 1997.

- [22] Christopher Kruegel and Giovanni Vigna. Anomaly detection of Web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*, New York, NY, October 2003.
- [23] Lancop StealthWatch. <http://www.lancop.com>.
- [24] Robert Lemos. Microsoft patch opens users to attack. <http://www.securityfocus.com/news/11408>, August 2006.
- [25] LinkScanner Pro. <http://www.explabs.com/products/lsprow.asp>.
- [26] Dahlia Malkhi and Michael K. Reiter. Secure execution of java applets using a remote playground. *IEEE Transactions on Software Engineering*, 26(12):1197–1209, 2000.
- [27] Mikhail Mikhailov and Craig Wills. Embedded objects in Web pages. Technical Report WPI-CS-TR-0005, Worcester Polytechnic Institute, Worcester, MA, March 2000.
- [28] Alexander Moshchuk, Tanya Bragin, Steven Gribble, and Henry Levy. A crawler-based study of spyware on the Web. In *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS '06)*, San Diego, CA, February 2006.
- [29] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS '05)*, San Diego, CA, February 2005.
- [30] Niels Provos. SpyBye. <http://www.spybye.org>.
- [31] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.
- [32] Prevx. <http://www.prevx.com>.
- [33] Charles Reis, John Dunagan, Helen Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.
- [34] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
- [35] Sana Security. <http://www.sanasecurity.com>.
- [36] Sandboxie. <http://www.sandboxie.com>.
- [37] Stefan Saroiu, Steven Gribble, and Henry Levy. Measurement and analysis of spyware in a university environment. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, March 2004.
- [38] Stefan Saroiu, Krishna Gummadi, Richard Dunn, Steven Gribble, and Henry Levy. An analysis of internet content delivery systems. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, New York, NY, December 2002. ACM Press.
- [39] Prabhat Singh and Arun Lakhota. Analysis and detection of computer viruses and worms: An annotated bibliography. *ACM SIGPLAN Notices*, 37(2):29–35, February 2002.
- [40] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, December 2004.
- [41] SiteAdvisor, Inc. <http://www.siteadvisor.com>.
- [42] Snort. The open source network intrusion detection system. <http://www.snort.org>.
- [43] StopBadware. <http://www.stopbadware.org/>.
- [44] StopBadware.org - Incompetence or McCarthyism 2.0? <http://www.adwarereport.com/mt/archives/stopbadwareorg.php>.
- [45] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 Annual USENIX Technical Conference*, Boston, MA, June 2001.
- [46] Hao Wang, Somesh Jha, and Vinod Ganapathy. NetSpy: Automatic generation of spyware signatures for NIDS. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*, Miami Beach, FL, December 2006. <http://dx.doi.org/10.1109/ACSAC.2006.34>.
- [47] Helen Wang, Chuanxiong Guo, Daniel Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM 2004*, Portland, OR, August 2004.
- [48] XiaoFeng Wang, Zhuowei Li, Jun Xu, Michael K. Reiter, Chongkyung Kil, and Jong Youl Choi. Packet vaccine: Black-box exploit detection and signature generation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS '06)*, October 2006.
- [49] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Samuel T. King. Automated Web patrol with Strider HoneyMonkeys: Finding Web sites that exploit browser vulnerabilities. In *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS '06)*, San Diego, CA, February 2006.
- [50] Yi-Min Wang, Doug Beck, Binh Vo, Roussi Roussev, Chad Verbowski, and Aaron Johnson. Detecting stealth software with Strider GhostBuster. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN '05)*, Yokohama, Japan, July 2005.
- [51] Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, Ming-Wei Wu, Yennun Huang, and Sy-Yen Kuo. Gatekeeper: Monitoring auto-start extensibility points (ASEPs) for spyware management. In *Proceedings of 18th Large Installation System Administration Conference (LISA '04)*, Atlanta, GA, November 2004.

- [52] Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Molly Brown, Tashana Landray, Denise Pinnel, Anna Karlin, and Henry Levy. Organization-based analysis of Web-object sharing and caching. In *Proceedings of the 2nd USENIX Conference on Internet Technologies and Systems (USITS '99)*, Boulder, CO, October 1999.
- [53] Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry Levy. On the scale and performance of cooperative Web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, SC, December 1999.

# Language Identification of Encrypted VoIP Traffic: *Alejandra y Roberto or Alice and Bob?*

Charles V. Wright

Lucas Ballard

Fabian Monroe

Gerald M. Masson

{cvwright, lucas, fabian, masson}@jhu.edu

Department of Computer Science

Johns Hopkins University

Baltimore, MD, USA

## Abstract

Voice over IP (VoIP) has become a popular protocol for making phone calls over the Internet. Due to the potential transit of sensitive conversations over untrusted network infrastructure, it is well understood that the contents of a VoIP session should be encrypted. However, we demonstrate that current cryptographic techniques do not provide adequate protection when the underlying audio is encoded using bandwidth-saving Variable Bit Rate (VBR) coders. Explicitly, we use the length of encrypted VoIP packets to tackle the challenging task of identifying the language of the conversation. Our empirical analysis of 2,066 native speakers of 21 different languages shows that a substantial amount of information can be discerned from encrypted VoIP traffic. For instance, our 21-way classifier achieves 66% accuracy, almost a 14-fold improvement over random guessing. For 14 of the 21 languages, the accuracy is greater than 90%. We achieve an overall binary classification (e.g., “*Is this a Spanish or English conversation?*”) rate of 86.6%. Our analysis highlights what we believe to be interesting new privacy issues in VoIP.

## 1 Introduction

Over the last several years, Voice over IP (VoIP) has enjoyed a marked increase in popularity, particularly as a replacement of traditional telephony for international calls. At the same time, the security and privacy implications of conducting everyday voice communications over the Internet are not yet well understood. For the most part, the current focus on VoIP security has centered around efficient techniques for ensuring confidentiality of VoIP conversations [3, 6, 14, 37]. Today, because of the success of these efforts and the attention they have received, it is now widely accepted that VoIP traffic should be encrypted before transmission over the Internet. Nevertheless, little, if any, work has explored the threat of

traffic analysis of encrypted VoIP calls. In this paper, we show that although encryption prevents an eavesdropper from reading packet contents and thereby listening in on VoIP conversations (for example, using [21]), traffic analysis can still be used to infer more information than expected—namely, the spoken language of the conversation. Identifying the spoken language in VoIP communications has several obvious applications, many of which have substantial privacy ramifications [7].

The type of traffic analysis we demonstrate in this paper is made possible because current recommendations for encrypting VoIP traffic (generally, the application of length-preserving stream ciphers) do not conceal the size of the plaintext messages. While leaking message size may not pose a significant risk for more traditional forms of electronic communication such as email, properties of real-time streaming media like VoIP greatly increase the potential for an attacker to extract meaningful information from plaintext length. For instance, the size of an encoded audio frame may have much more meaningful semantics than the size of a text document. Consequently, while the size of an email message likely carries little information about its contents, the use of bandwidth-saving techniques such as variable bit rate (VBR) coding means that the size of a VoIP packet is directly determined by the type of sound its payload encodes. This information leakage is exacerbated in VoIP by the sheer number of packets that are sent, often on the order of tens or hundreds every second. Access to such large volumes of packets over a short period of time allows an adversary to quickly estimate meaningful distributions over the packet lengths, and in turn, to learn information about the language being spoken.

Identifying spoken languages is a task that, on the surface, may seem simple. However it is a problem that has not only received substantial attention in the speech and natural language processing community, but has also been found to be challenging even with access to *full* acoustic data. Our results show an encrypted conversa-

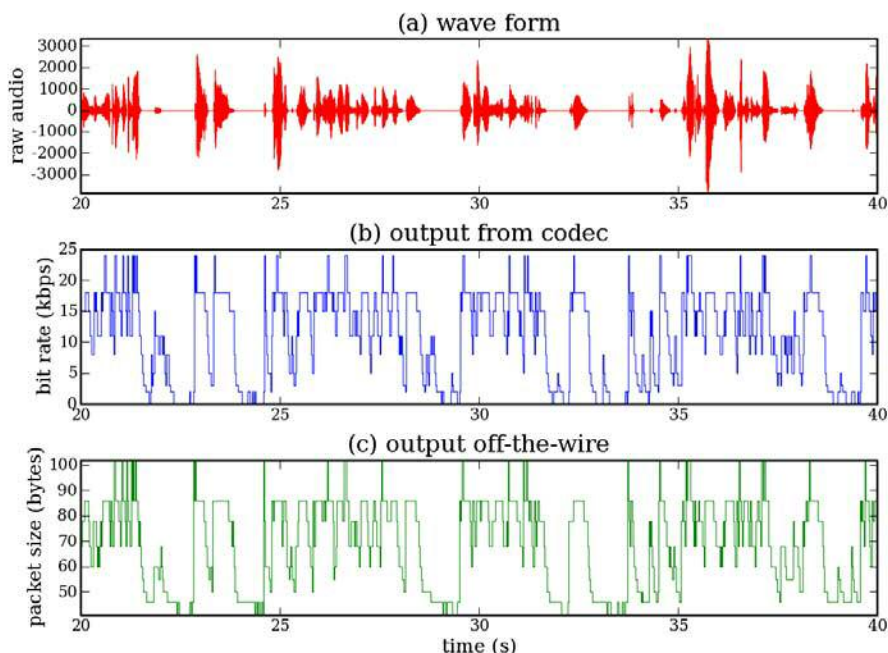


Figure 1: Uncompressed audio signal, Speex bit rates, and packet sizes for a random sample from the corpus.

tion over VoIP can leak information about its contents, to the extent that an eavesdropper can successfully infer what language is being spoken. The fact that VoIP packet lengths can be used to perform any sort of language identification is interesting in and of itself. Our success with language identification in this setting provides strong grounding for mandating the use of fixed length compression techniques in VoIP, or for requiring the underlying cryptographic engine to pad each packet to a common length.

The rest of this paper is organized as follows. We begin in Section 2 by reviewing why and how voice over IP technologies leak information about the language spoken in an encrypted call. In Section 3, we describe our design for a classifier that exploits this information leakage to automatically identify languages based on packet sizes. We evaluate this classifier’s effectiveness in Section 4, using open source VoIP software and audio samples from a standard data set used in the speech processing community. We review related work on VoIP security and information leakage attacks in Section 5, and conclude in Section 6.

## 2 Information Leakage via Variable Bit Rate Encoding

To highlight why language identification is possible in encrypted VoIP streams, we find it instructive to first re-

view the relevant inner workings of a modern VoIP system. Most VoIP calls use at least two protocols: (1) a signaling protocol such as the Session Initiation Protocol (SIP) [23] used for locating the callee and establishing the call and (2) the Real Time Transport Protocol (RTP) [25, 4] which transmits the audio data, encoded using a special-purpose speech codec, over UDP. While several speech codecs are available (including G.711 [10], G.729 [12], Speex [29], and iLBC [2]), we choose the Speex codec for our investigation as it offers several advanced features like a VBR mode and discontinuous transmission, and its source code is freely available. Additionally, although Speex is not the only codec to offer variable bit rate encoding for speech [30, 16, 20, 35, 5], it is the most popular of those that do.

Speex, like most other modern speech codecs, is based on code-excited linear prediction (CELP) [24]. In CELP, the encoder uses vector quantization with both a fixed codebook and an adaptive codebook [22] to encode a window of  $n$  audio samples as one frame. For example, in the Speex default narrowband mode, the audio input is sampled at 8kHz, and the frames each encode 160 samples from the source waveform. Hence, a packet containing one Speex frame is typically transmitted every 20ms. In VBR mode, the encoder takes advantage of the fact that some sounds are easier to represent than others. For example, with Speex, vowels and high-energy transients

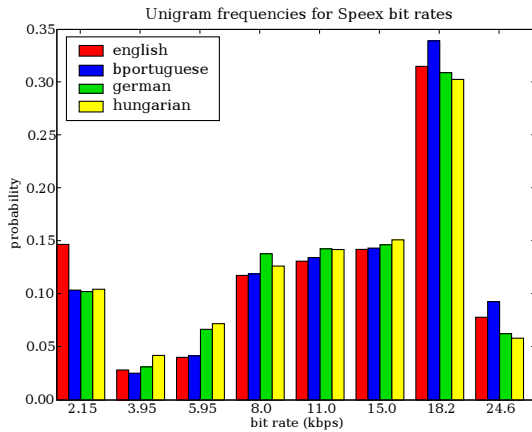


Figure 2: Unigram frequencies of bit rates for English, Brazilian Portuguese, German and Hungarian

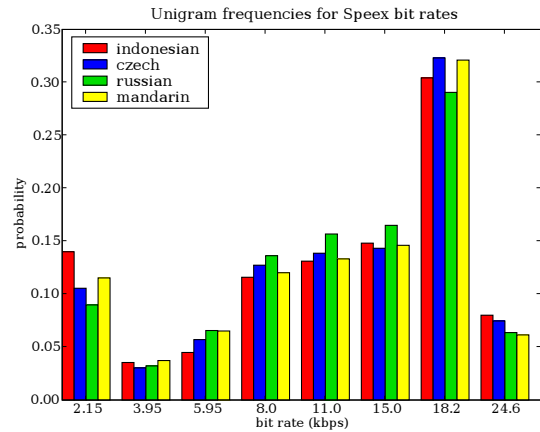


Figure 3: Unigram frequencies of bit rates for Indonesian, Czech, Russian, and Mandarin

require higher bit rates than fricative sounds like “s” or “f” [28]. To achieve improved sound quality and a low (average) bit rate, the encoder uses fewer bits to encode frames which contain “easy” sounds and more bits for frames with sounds that are harder to encode. Because the VBR encoder selects the best bit rate for each frame, the size of a packet can be used as a predictor of the bit rate used to encode the corresponding frame. Therefore, given only packet lengths, it is possible to extract information about the underlying speech. Figure 1, for example, shows an audio input, the encoder’s bit rate, and the resulting packet sizes as the information is sent on the wire; notice how strikingly similar the last two cases are.

As discussed earlier, by now it is commonly accepted that VoIP traffic should not be transmitted over the Internet without some additional security layer [14, 21]. Indeed, a number of proposals for securing VoIP have already been introduced. One such proposal calls for tunneling VoIP over IPSec, but doing so imposes unacceptable delays on a real-time protocol [3]. An alternative, endorsed by NIST [14], is the Secure Real Time Transport Protocol (SRTP) [4]. SRTP is an extension to RTP and provides confidentiality, authenticity, and integrity for real-time applications. SRTP allows for three modes of encryption: AES in counter mode, AES in f8-mode, and no encryption. For the two stream ciphers, the standard states that “in case the payload size is not an integer multiple of (the block length), the excess bits of the key stream are simply discarded” [4]. Moreover, while the standard permits higher level protocols to pad their messages, the default in SRTP is to use length-preserving encryption and so one can still derive information about the underlying speech by observing the lengths of the encrypted payloads.

Given that the sizes of encrypted payloads are closely

related to bit rates used by the speech encoder, a pertinent question is whether different languages are encoded at different bit rates. Our conjecture is that this is indeed the case, and to test this hypothesis we examine real speech data from the Oregon Graduate Institute Center for Speech Learning and Understanding’s “22 Language” telephone speech corpus [15]. The data set consists of speech from native speakers of 21 languages, recorded over a standard telephone line at 8kHz. This is the same sampling rate used by the Speex narrowband mode. General statistics about the data set are provided in Appendix A.

As a preliminary test of our hypothesis, we encoded all of the audio files from the CSLU corpus and recorded the sequence of bit rates used by Speex for each file. In narrowband VBR mode with discontinuous transmission enabled, Speex encodes the data set using nine distinct bit rates, ranging from 0.25kbps up to 24.6kbps. Figure 2 shows the frequency for each bit rate for English, Brazilian Portuguese, German, and Hungarian. For most bit rates, the frequencies for English are quite close to those for Portuguese; but Portuguese and Hungarian appear to exhibit different distributions. This results suggest that distinguishing Portuguese from Hungarian, for example, would be less challenging than differentiating Portuguese from English, or Indonesian from Russian (see Figure 3).

Figures 4 and 5 provide additional evidence that bigram frequencies (i.e., the number of instances of consecutively observed bit rate pairs) differ between languages. The  $x$  and  $y$  axes of both figures specify observed bit rates. The density of the square  $(x, y)$  shows the difference in probability of bigram  $x, y$  between the two languages divided by the average probability of bigram  $x, y$  between the two. Thus, dark squares indicate

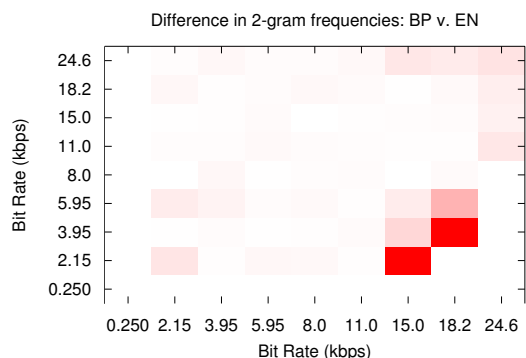


Figure 4: The normalized difference in bigram frequencies between Brazilian Portuguese (BP) and English (EN).

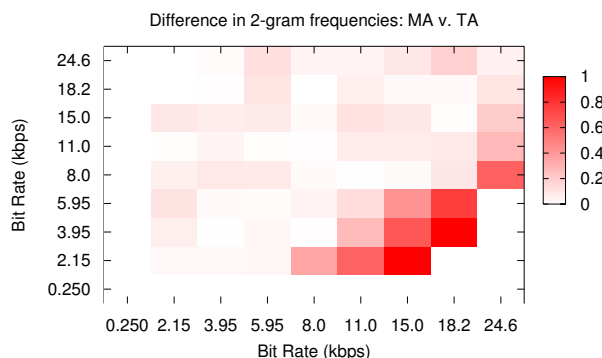


Figure 5: The normalized difference in bigram frequencies between Mandarin (MA) and Tamil (TA).

significant differences between the languages for an observed bigram. Notice that while Brazilian Portuguese (BP) and English (EN) are similar, there are differences between their distributions (see Figure 4). Languages such as Mandarin (MA) and Tamil (TA) (see Figure 5), exhibit more substantial incongruities.

Encouraged by these results, we applied the  $\chi^2$  test to examine the similarity between sample unigram distributions. The  $\chi^2$  test is a non-parametric test that provides an indication as to the likelihood that samples are drawn from the same distribution. The  $\chi^2$  results confirmed (with high confidence) that samples from the same language have similar distributions, while those from different languages do not. In the next section, we explore techniques for exploiting these differences to automatically identify the language spoken in short clips of encrypted VoIP streams.

### 3 Classifier

We explored several classifiers (e.g., using techniques based on  $k$ -Nearest Neighbors, Hidden Markov Models, and Gaussian Mixture Models), and found that a variant of a  $\chi^2$  classifier provided a similar level of accuracy, but was more computationally efficient. In short, the  $\chi^2$  classifier takes a set of samples from a speaker and models (or probability distributions) for each language, and classifies a speaker as belonging to the language for which the  $\chi^2$  distance between the speaker's model and the language's model is minimized. To construct a language model, each speech sample (i.e., a phone call), is represented as a series of packet lengths generated by a Speex-enabled VoIP program. We simply count the  $n$ -grams of packet lengths in each sample to estimate the multinomial distribution for that model (for our empiri-

cal analysis, we set  $n = 3$ ). For example, if given a stream of packets with lengths of 55, 86, 60, 50 and 46 bytes, we would extract the 3-grams (55, 86, 60), (86, 60, 50), (60, 50, 46), and use those triples to estimate the distributions<sup>1</sup>. We do not distinguish whether a packet represents speech or silence (as it is difficult to do so with high accuracy), and simply count each  $n$ -gram in the stream.

It is certainly the case that some  $n$ -grams will be more useful than others for the purposes of language separation. To address this, we modify the above construction such that our models only incorporate  $n$ -grams that exhibit low intraclass variance (i.e., the speakers within the same language exhibit similar distributions on the  $n$ -gram of concern) and high interclass variance (i.e., the speakers of one language have different distributions than those of other languages for that particular  $n$ -gram). Before explaining how to determine the distinguishability of a  $n$ -gram  $g$ , we first introduce some notation. Assume we are given a set of languages,  $\mathcal{L}$ . Let  $P_L(g)$  denote the probability of the  $n$ -gram  $g$  given the language  $L \in \mathcal{L}$ , and  $P_s(g)$  denote the probability of the  $n$ -gram  $g$  given the speaker  $s \in L$ . All probabilities are estimated by dividing the total number of occurrences of a given  $n$ -gram by the total number of observed  $n$ -grams.

For the  $n$ -gram  $g$  we compute its average intraclass variability as:

$$\text{VAR}_{\text{intra}}(g) = \frac{1}{|\mathcal{L}|} \sum_{L \in \mathcal{L}} \frac{1}{|L|} \sum_{s \in L} (P_s(g) - P_L(g))^2$$

Intuitively, this measures the average distance between the probability of  $g$  for given a speaker and the probability of  $g$  given that speaker's language; i.e., the average variance of the probability distributions  $P_L(g)$ . We com-

pute the interclass variability as:

$$\text{VAR}_{\text{inter}}(g) = \left( (|\mathcal{L}| - 1) \sum_{L \in \mathcal{L}} |L| \right)^{-1} \cdot \left( \sum_{L_1 \in \mathcal{L}} \sum_{s \in L_1} \sum_{L_2 \in \mathcal{L} \setminus L_1} (P_s(g) - P_{L_2}(g))^2 \right)$$

This measures, on average, the difference between the probability of  $g$  for a given speaker and the probability of  $g$  given every other language. The second two summations in the second term measure the distance from each speaker in a specific language to the means of all other languages. The first summation and the leading normalization term are used to compute the average over all languages. As an example, if we consider the seventh and eighth bins in the unigram case illustrated in Figure 2, then  $\text{VAR}_{\text{inter}}(15.0 \text{ kbps}) < \text{VAR}_{\text{inter}}(18.2 \text{ kbps})$ .

We set the overall distinguishability for  $n$ -gram  $g$  to be  $\text{DIS}(g) = \text{VAR}_{\text{inter}}(g)/\text{VAR}_{\text{intra}}(g)$ . Intuitively, if  $\text{DIS}(g)$  is large, then speakers of the same language tend to have similar probability densities for  $g$ , and these densities will vary across languages. We choose to make our classification decisions using only those  $g$  with  $\text{DIS}(g) > 1$ , we denote this set of distinguishing  $n$ -grams as  $G$ . The model for language  $L$  is simply the probability distribution  $P_L$  over  $G$ .

To further refine the models, we remove outliers (speakers) who might contribute noise to each distribution. In order to do this, we must first specify a distance metric between a speaker  $s$  and a language  $L$ . Suppose that we extract  $N$  total  $n$ -grams from  $s$ 's speech samples. Then, we compute the distance between  $s$  and  $L$  as:

$$\Delta(P_s, P_L, G) = \sum_{g \in G} \frac{(N \cdot P_L(g) - N \cdot P_s(g))^2}{N \cdot P_L(g)}$$

We then remove the speakers  $s$  from  $L$  for which  $\Delta(P_s, P_L, G)$  is greater than some language-specific threshold  $t_L$ . After we have removed these outliers, we recompute  $P_L$  with the remaining speakers.

Given our refined models, our goal is to use a speaker's samples to identify the speaker's language. We assign the speaker  $s$  to the language with the model that is closest to the speaker's distribution over  $G$  as follows:

$$L^* = \underset{L \in \mathcal{L}}{\text{argmin}} \Delta(P_s, P_L, G)$$

To determine the accuracy of our classifier, we apply the standard leave-one-out cross validation analysis to each speaker in our data set. That is, for a given speaker, we remove that speaker's samples and use the remaining

samples to compute  $G$  and the models  $P_L$  for each language in  $L \in \mathcal{L}$ . We choose the  $t_L$  such that 15% of the speakers are removed as outliers (these outliers are eliminated during model creation, but they are still included in classification results). Next, we compute the probability distribution,  $P_s$ , over  $G$  using the speaker's samples. Finally, we classify the speaker using  $P_s$  and the outlier-reduced models derived from the other speakers in the corpus.

## 4 Empirical Evaluation

To evaluate the performance of our classifier in a realistic environment, we simulated VoIP calls for many different languages by playing audio files from the Oregon Graduate Institute Center for Speech Learning & Understanding's "22 Language" telephone speech corpus [15] over a VoIP connection. This corpus is widely used in language identification studies in the speech recognition community (e.g. [19], [33]). It contains recordings from a total of 2066 native speakers of 21 languages<sup>2</sup>, with over 3 minutes of audio per speaker. The data was originally collected by having users call in to an automated telephone system that prompted them to speak about several topics and recorded their responses. There are several files for each user. In some, the user was asked to answer a question such as "Describe your most recent meal" or "What is your address?" In others, they were prompted to speak freely for up to one minute. This type of free-form speech is especially appealing for our evaluation because it more accurately represents the type of speech that would occur in a real telephone conversation. In other files, the user was prompted to speak in English or was asked about the language(s) they speak. To avoid any bias in our results, we omit these files from our analysis, leaving over 2 minutes of audio for each user. See Appendix A for specifics concerning the dataset.

Our experimental setup includes two PC's running Linux with open source VoIP software [17]. One of the machines acts as a server and listens on the network for SIP calls. Upon receiving a call, it automatically answers and negotiates the setup of the voice channel using Speex over RTP. When the voice channel is established, the server plays a file from the corpus over the connection to the caller, and then terminates the connection. The caller, which is another machine on our LAN, automatically dials the SIP address of the server and then "listens" to the file the server plays, while recording the sequence of packets sent from the server. The experimental setup is depicted in Figure 6.

Although our current evaluation is based on data collected on a local area network, we believe that languages could be identified under most or all network conditions where VoIP is practical. First, RTP (and SRTP) sends in

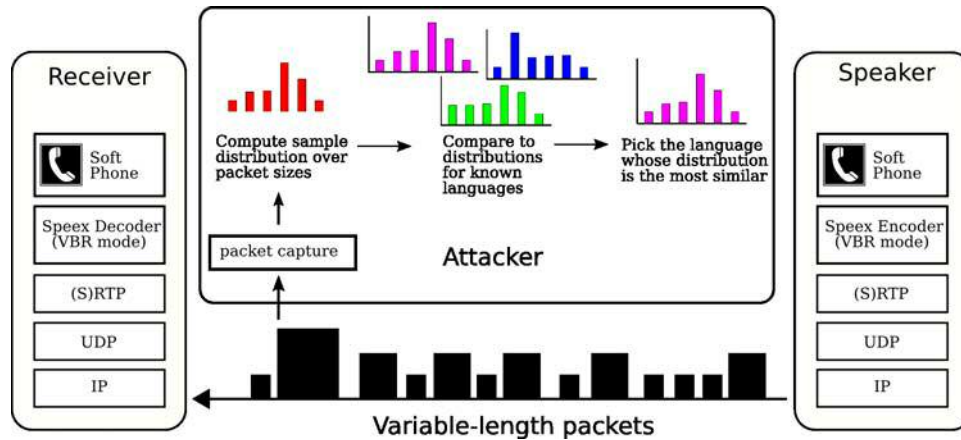


Figure 6: Experimental setup.

the clear a timestamp corresponding to the sampling time of the first byte in the packet data [25]. This timestamp can therefore be used to infer packet ordering and identify packet loss. Second, VoIP is known to degrade significantly under undesirable network connections with latency more than a few hundred milliseconds [11], and it is also sensitive to packet loss [13]. Therefore any network which allows for acceptable call quality should also give our classifier a sufficient number of trigrams to make an accurate classification.

For a concrete test of our techniques on wide-area network data, we performed a smaller version of the above experiment by playing a reduced set of 6 languages across the Internet between a server on our LAN and a client machine on a residential DSL connection. In the WAN traces, we observed less than 1% packet loss, and there was no statistically significant difference in recognition rates for the LAN and WAN experiments.

#### 4.1 Classifier Accuracy

In what follows, we examine the classifier’s performance when trained using all available samples (excluding, of course, the target user’s samples). To do so, we test each speaker against all 21 models. The results are presented in Figures 7 and 8. Figure 7 shows the confusion matrix resulting from the tests. The  $x$  axis specifies the language of the speaker, and the  $y$  axis specifies the language of the model. The density of the square at position  $(x, y)$  indicates how often samples from speakers of language  $x$  were classified as belonging to language  $y$ .

To grasp the significance of our results, it is important to note that if packet lengths leaked no information, then the classification rates for each language would be close to random, or about 4.8%. However, the confusion matrix shows a general density along the  $y = x$  line. The classifier performed best on Indonesian (IN) which

is accurately classified 40% of the time (an eight fold improvement over random guessing). It also performed well on Russian (RU), Tamil (TA), Hindi (HI), and Korean (KO), classifying at rates of 35, 35, 29 and 25 percent, respectively. Of course, Figure 7 also shows that in several instances, misclassification occurs. For instance, as noted in Figure 2, English (EN) and Brazilian Portuguese (BP) exhibit similar unigram distributions, and indeed when misclassified, English was often confused with Brazilian Portuguese (14% of the time). Nonetheless, we believe these results are noteworthy, as if VoIP did not leak information, the classification rates would be close to those of random guessing. Clearly, this is not the case, and our overall accuracy was 16.3%—that is, a three and a half fold improvement over random guessing.

An alternative perspective is given in Figure 8, which shows how often the speaker’s language was among the classifier’s top  $x$  choices. We plot random guessing as a baseline, along with languages that exhibited the highest and lowest classification rates. On average, the correct language was among our top four speculations 50.2% of the time. Note the significant improvement over random guessing, which would only place the correct language in the top four choices approximately 19% of the time. Indonesian is correctly classified in our top three choices 57% of the time, and even Arabic—the language with the lowest overall classification rates—was correctly placed among our top three choices 30% of the time.

In many cases, it might be worthwhile to distinguish between only two languages, e.g., whether an encrypted conversation in English or Spanish. We performed tests that aimed at identifying the correct language when supplied only two possible choices. We see a stark improvement over random guessing, with seventy-five percent of the language combinations correctly distinguished with an accuracy greater than 70.1%; twenty-five percent had accuracies greater than 80%. Our overall binary classifi-

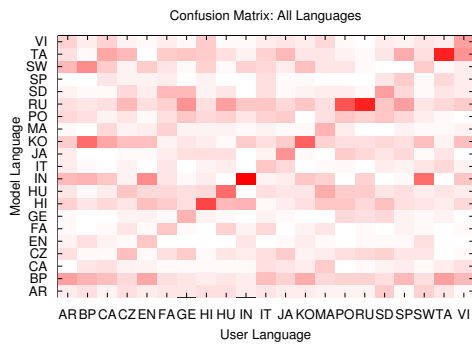


Figure 7: Confusion Matrix for 21-way test using trigrams. Darkest and lightest boxes represent accuracies of 0.4 and 0.0, respectively.

cation rate was 75.1%.

Our initial intuition in Section 2 are strongly correlated to our empirical results. For example, rates for Russian versus Italian and Mandarin versus Tamil (see Figure 5) were 78.5% and 84%, respectively. The differences in the histograms shown earlier (Figure 2) also have direct implications for our classification rates in this case. For instance, our classifier’s accuracy when tasked with distinguishing between Brazilian Portuguese and English was only 66.5%, whereas the accuracy for English versus Hungarian was 86%.

## 4.2 Reducing Dimensionality to Improve Performance

Although these results adequately demonstrate that length-preserving encryption leaks information in VoIP, there are limiting factors to the aforementioned approach that hinder classification accuracy. The primary difficulty arises from the fact that the classifier represents each speaker and language as a probability distribution over a very high dimensional space. Given 9 different observed packet lengths, there are 729 possible different trigrams. Of these possibilities, there are 451 trigrams that are useful for classification, i.e.,  $\text{DIS}(g) > 1$  (see Section 3). Thus, speaker and language models are probability distributions over a 451-dimensional space. Unfortunately, given our current data set of approximately 7,277 trigrams per speaker, it is difficult to estimate densities over such a large space with high precision.

One way to address this problem is based on the observation that some bit rates are used in similar ways by the Speex encoder. For example, the two lowest bit rates, which result in packets of 41 and 46 bytes, respectively, are often used to encode periods of silence

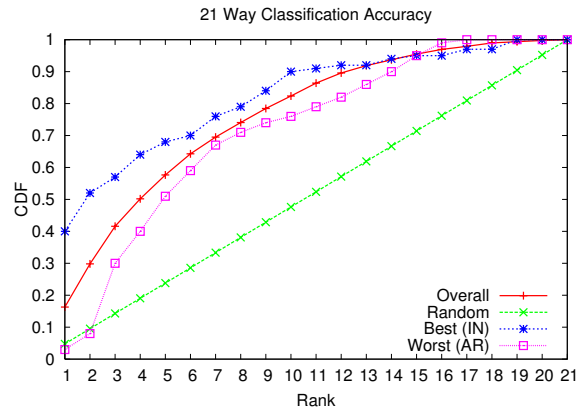


Figure 8: CDF showing how often the speaker’s language was among the classifier’s top  $x$  choices.

or non-speech. Therefore, we can reasonably consider the two smallest packet sizes functionally equivalent and put them together into a single group. In the same way, other packet sizes may be used similarly enough to warrant grouping them together as well. We experimented with several mappings of packet sizes to groups, but found that the strongest results are obtained by mapping the two smallest packet lengths together, mapping all of the mid-range packet lengths together, and leaving the largest packet size in a group by itself.

We assign each group a specific symbol,  $s$ , and then compute  $n$ -grams from these symbols instead of the original packet sizes. So, for example, given the sequence of packet lengths 41, 50, 46, and 55, we map 41 and 46 to  $s_1$  and 50 and 55 to  $s_2$  to extract the 3-grams  $(s_1, s_2, s_1)$  and  $(s_2, s_1, s_2)$ , etc. Our classification process then continues as before, except that the reduction in the number of symbols allows us to expand our analysis to 4-grams. After removing the 4-grams  $g$  with  $\text{DIS}(g) < 1$ , we are left with 47 different 4-gram combinations. Thus, we reduced the dimensionality of the points from 451 to 47. Here we are estimating distributions over a 47-dimensional space using on average of 7,258 4-grams per speaker.

Results for this classifier are shown in Figures 9 and 10. With these improvements, the 21-way classifier correctly identifies the language spoken 66% of the time—a fourfold improvement over our original classifier and more than 13 times better than random guessing. It recognizes 14 of the 21 languages exceptionally well, identifying them with over 90% accuracy. At the same time, there is a small group of languages which the new classifier is not able to identify reliably; Czech, Spanish, and Vietnamese are never identified correctly on the first try. This occurs mainly because the languages which are not

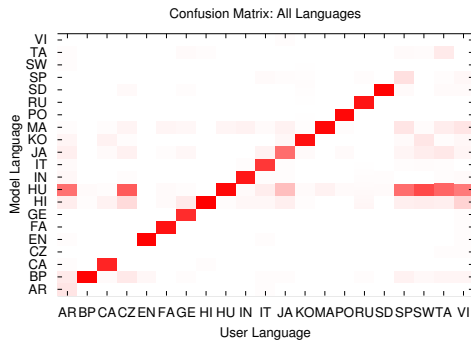


Figure 9: Confusion Matrix for the 21-way test using 4-grams and reduced set of symbols. Darkest and lightest boxes represent accuracies of 1.0 and 0.0, respectively.

recognized accurately are often misidentified as one of a handful of other languages. Hungarian, in particular, has false positives on speakers of Arabic, Czech, Spanish, Swahili, Tamil, and Vietnamese. These same languages are also less frequently misidentified as Brazilian Portuguese, Hindi, Japanese, Korean, or Mandarin. In future work, we plan to investigate what specific acoustic features of language cause this classifier to perform so well on many of the languages while failing to accurately recognize others.

Binary classification rates, shown in Figure 11 and Table 1, are similarly improved over our initial results. Overall, the classifier achieves over 86% accuracy when distinguishing between two languages. The median accuracy is 92.7% and 12% of the language pairs can be distinguished at rates greater than 98%. In a few cases like Portuguese versus Korean or Farsi versus Polish, the classifier exhibited 100% accuracy on our test data.

Interestingly, the results of our classifiers are comparable to those presented by Zissman [38] in an early study of language identification techniques using full acoustic data. Zissman implemented and compared four different language recognition techniques, including Gaussian mixture model (GMM) classification and techniques based on single-language phone recognition and  $n$ -gram language modeling. All four techniques used cepstral coefficients as input [22].

The GMM classifier described by Zissman is much simpler than the other techniques and serves primarily as a baseline for comparing the performance of the more sophisticated methods presented in that work. Its accuracy is quite close to that of our initial classifier: with access to approximately 10 seconds of raw acoustic data, it scored approximately 78% for three language pairs, compared to our classifier's 89%. The more sophisti-

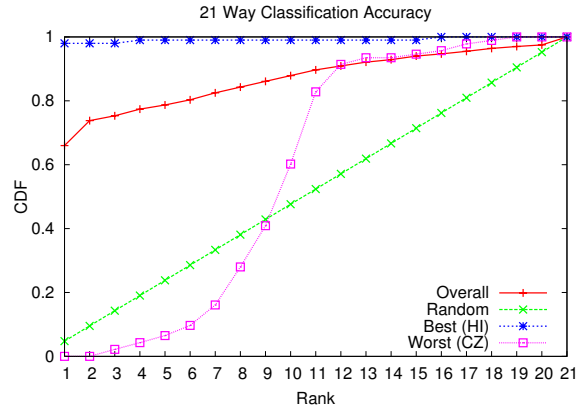


Figure 10: CDF showing how often the speaker's language was among the classifier's top  $x$  choices using 4-grams and reduced set of symbols.

cated classifiers in [38] have performance closer to that of our improved classifier. In particular, an 11-way classifier based on phoneme recognition and  $n$ -gram language modeling (PRLM) was shown to achieve 89% accuracy when given 45s of acoustic data. In each case, our classifier has the advantage of a larger sample, using around 2 minutes of data.

Naturally, current techniques for language identification have improved on the earlier work of Zissman and others, and modern error rates are almost an order of magnitude better than what our classifiers achieve. Nevertheless, this comparison serves to demonstrate the point that we are able to extract significant information from encrypted VoIP packets, and are able to do so with an accuracy close to a reasonable classifier with access to acoustic data.

## DISCUSSION

We note that since the audio files in our corpus were recorded over a standard telephone line, they are sampled at 8kHz and encoded as 16-bit PCM audio, which is appropriate for Speex narrowband mode. While almost all traditional telephony samples the source audio at 8kHz, many soft phones and VoIP codecs have the ability to use higher sampling rates such as 16kHz or 32kHz to achieve better audio quality at the tradeoff of greater load on the network. Unfortunately, without a higher-fidelity data set, we have been unable to evaluate our techniques on VoIP calls made with these higher sampling rates. Nevertheless, we feel that the results we derive from using the current training set are also informative for higher-bandwidth codecs for two reasons.

First, it is not uncommon for regular phone conversations to be converted to VoIP, enforcing the use of an

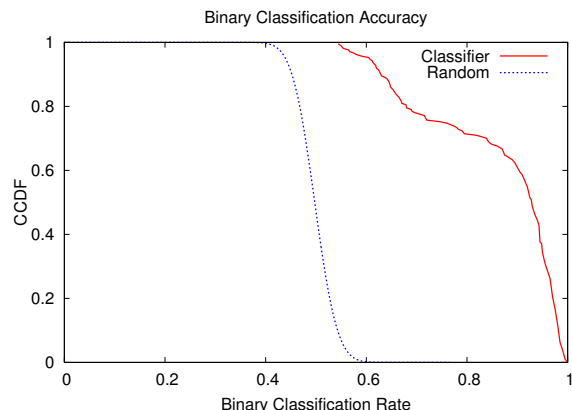


Figure 11: CCDF for overall accuracy of the binary classifier using 4-grams and reduced set of symbols.

8kHz sampling rate. Our test setup accurately models the traffic produced under this scenario. Second, and more importantly, by operating at the 8kHz level, we argue that we work with *less* information about the underlying speech, as we are only able to estimate bit rates up to a limited fidelity. Speex wideband mode, for example, operates on speech sampled at 16kHz and in VBR mode uses a wider range of bit rates than does the narrowband mode. With access to more distinct bit rates, one would expect to be able to extract more intricate characteristics about the underlying speech. In that regard, we believe that our results could be further improved given access to higher-fidelity samples.

### 4.3 Mitigation

Recall that these results are possible because the default mode of encryption in SRTP is to use a length-preserving stream cipher. However, the official standard [4] does allow implementations to optionally pad the plaintext payload to the next multiple of the cipher’s block size, so that the original payload size is obscured. Therefore, we investigate the effectiveness of padding against our attack, using several block sizes.

To determine the packet sizes that would be produced by encryption with padding, we simply modify the packet sizes we observed in our network traces by increasing their RTP payload sizes to the next multiple of the cipher’s block size. To see how our attack is affected by this padding, we re-ran our experiments using block sizes of 128, 192, 256, and 512 bits. Padding to a block size of 128 bits results in 4 distinct packet sizes; this number decreases to 3 distinct sizes with 192-bit blocks, 2 sizes with 256-bit blocks, and finally, with 512-bit blocks, all packets are the same size. Figure 12 shows the CDF for the classifier’s results for these four

Lang.	Acc.	Lang.	Acc
EN-FA	0.980	CZ-JA	0.544
GE-RU	0.985	AR-SW	0.549
FA-SD	0.990	CZ-HU	0.554
IN-PO	0.990	CZ-SD	0.554
PO-RU	0.990	MA-VI	0.565
BP-PO	0.995	JA-SW	0.566
EN-HI	0.995	HU-VI	0.575
HI-PO	0.995	CZ-MA	0.580
BP-KO	1.000	CZ-SW	0.590
FA-PO	1.000	HU-TA	0.605

Table 1: Binary classifier recognition rates for selected language pairs. Languages and their abbreviations are listed in Appendix A.

cases, compared to random guessing and to the results we achieve when there is no padding.

Padding to 128-bit blocks is largely ineffective because there is still sufficient granularity in the packet sizes that we can map them to basically to the same three bins used by our improved classifier in Section 4.2. Even with 192- or 256-bit blocks, where dimensionality reduction does not offer substantial improvement, the correct language can be identified on the first guess over 27% of the time—more than 5 times better than random guessing. It is apparent from these results that, for encryption with padding to be an effective defense against this type of information leakage, the block size must be large enough that all encrypted packets are the same size.

Relying on the cryptographic layer to protect against both eavesdropping and traffic analysis has a certain philosophical appeal because then the compression layer does not have to be concerned with security issues. On the other hand, padding incurs significant overhead in the number of bytes that must be transmitted. Table 2 lists the increase in traffic volume that arises from padding to each block size, as well as the improvement of the overall accuracy of the classifier over random guessing.

Another solution for ensuring that there is no information leakage is to use a constant bit rate codec, such as Speex in CBR mode, to send packets of fixed length. Forcing the encoder to use a fixed number of bits is an attractive approach, as the encoder could use the bits that would otherwise be used as padding to improve the quality of the encoded sound. While both of these approaches would detract from the bandwidth savings provided by VBR encoders, they provide much stronger privacy guarantees for the participants of a VoIP call.

Block Size	Overhead	Accuracy	Improvement vs Random
none	0.0%	66.0%	13.8x
128 bits	8.7%	62.5%	13.0x
192 bits	13.8%	27.1%	5.7x
256 bits	23.9%	27.2%	5.7x
512 bits	42.2%	6.9%	1.4x

Table 2: Tradeoff of effectiveness versus overhead incurred for padding VoIP packets to various block sizes.

## 5 Related Work

Some closely related work is that of Wang et al. [31] on tracking VoIP calls over low-latency anonymizing networks such as Tor [9]. Unlike our analysis, which is entirely passive, the attack in [31] requires that the attacker be able to *actively* inject delays into the stream of packets as they traverse the anonymized network. Other recent work has explored extracting sensitive information from several different kinds of encrypted network connections. Sun et al. [27], for example, examined World Wide Web traffic transmitted in HTTP over secure (SSL) connections and were able to identify a set of sensitive websites based on the number and sizes of objects in each encrypted HTTP response. Song et al. [26] used packet interarrival times to infer keystroke patterns and ultimately crack passwords typed over SSH. Zhang and Paxson [36] also used packet timing in SSH traffic to identify pairs of connections which form part of a chain of “stepping stone” hosts between the attacker and his eventual victim. In addition to these application-specific attacks, our own previous work demonstrates that packet size and timing are indicative of the application protocol used in SSL-encrypted TCP connections and in simple forms of encrypted tunnels [34].

Techniques for automatically identifying spoken languages were the subject of a great deal of work in the mid 1990’s [18, 38]. While these works used a wide range of features extracted from the audio data and employed many different machine learning techniques, they all represent attempts to mimic the way humans differentiate between languages, based on differences in the sounds produced. Because our classifier does not have direct access to the acoustic data, it is unrealistic to expect that it could outperform a modern language recognition system, where error rates in the single digits are

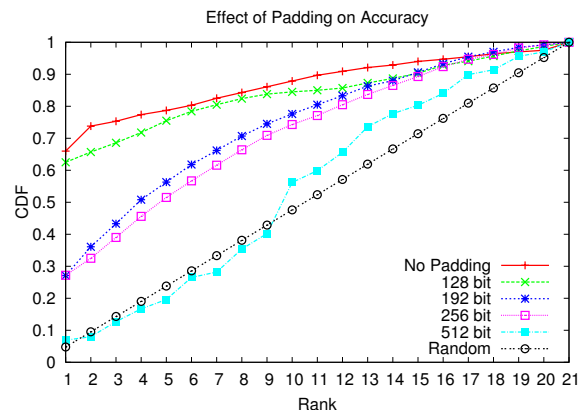


Figure 12: The effect of padding on classifier accuracy.

not uncommon. Nevertheless, automatic language identification is not considered a solved problem, even with access to full acoustic data, and work is ongoing in the speech community to improve recognition rates and explore new approaches (see, e.g., [32, 8, 1]).

## 6 Conclusions

In this paper, we show that despite efforts devoted to securing conversations that traverse Voice over IP, an adversary can still exploit packet lengths to discern considerable information about the underlying spoken language. Our techniques examine patterns in the output of Variable Bit Rate encoders to infer characteristics of the encoded speech. Using these characteristics, we evaluate our techniques on a large corpus of traffic from different speakers, and show that our techniques can classify (with reasonable accuracy) the language of the target speaker. Of the 21 languages we evaluated, we are able to correctly identify 14 with accuracy greater than 90%. When tasked with distinguishing between just two languages, our average accuracy over all language pairs is greater than 86%. These recognition rates are on par with early results from the language identification community, and they demonstrate that variable bit rate coding leaks significant information. Moreover, we show that simple padding is insufficient to prevent leakage of information about the language spoken. We believe that this information leakage from encrypted VoIP packets is a significant privacy concern. Fortunately, we are able to suggest simple remedies that would thwart our attacks.

## Acknowledgments

We thank Scott Coull for helpful conversations throughout the course of this research, as well as for pointing out

the linphone application [17]. We also thank Patrick McDaniel and Patrick Traynor for their insightful comments on early versions of this work. This work was funded in part by NSF grants CNS-0546350 and CNS-0430338.

## Notes

<sup>1</sup>Note that our classifier is not a true instance of a  $\chi^2$  classifier as the probability distributions over each  $n$ -gram are not independent. Essentially, we just use the  $\chi^2$  function as a multi-dimensional distance metric.

<sup>2</sup>Due to problems with the data, recordings from the French speakers are unavailable.

## References

- [1] NIST language recognition evaluation. <http://www.nist.gov/speech/tests/lang/index.htm>.
- [2] S. Andersen, A. Duric, H. Astrom, R. Hagen, W. Kleijn, and J. Linden. Internet Low Bit Rate Codec (iLBC), 2004. RFC 3951.
- [3] R. Barbieri, D. Bruschi, and E. Rosti. Voice over IPsec: Analysis and solutions. In *Proceedings of the 18th Annual Computer Security Applications Conference*, pages 261–270, December 2002.
- [4] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norman. The secure real-time transport protocol (SRTP). RFC 3711.
- [5] F. Beritelli. High quality multi-rate CELP speech coding for wireless ATM networks. In *Proceedings of the 1998 Global Telecommunications Conference*, volume 3, pages 1350–1355, November 1998.
- [6] P. Biondi and F. Desclaux. Silver needle in the Skype. In *BlackHat Europe*, 2006. <http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-biondi/bh-eu-06-biondi-up.pdf>.
- [7] M. Blaze. Protocol failure in the escrowed encryption standard. In *Proceedings of Second ACM Conference on Computer and Communications Security*, pages 59–67, 1994.
- [8] L. Burget, P. Matejka, and J. Cernocky. Discriminative training techniques for acoustic language identification. In *Proceedings of the 2006 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages I–209–I–212, May 2006.
- [9] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320, August 2004.
- [10] International Telecommunications Union. Recommendation G.711: Pulse code modulation (PCM) of voice frequencies, 1988.
- [11] International Telecommunications Union. Recommendation P.1010: Fundamental voice transmission objectives for VoIP terminals and gateways, 2004.
- [12] International Telecommunications Union. Recommendation G.729: Coding of speech at 8 kbits using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP), 2007.
- [13] W. Jiang and H. Schulzrinne. Modeling of packet loss and delay and their effect on real-time multimedia service quality. In *Proceedings of the 10th International Workshop on Network and Operating System Support for Digital Audio and Video*, June 2000.
- [14] D. R. Kuhn, T. J. Walsh, and S. Fries. Security considerations for voice over IP systems. Technical Report Special Publication 008-58, NIST, January 2005.
- [15] T. Lander, R. A. Cole, B. T. Oshika, and M. Noel. The OGI 22 language telephone speech corpus. In *EUROSPEECH*, pages 817–820, 1995.
- [16] S. McClellan and J. D. Gibson. Variable-rate CELP based on subband flatness. *IEEE Transactions on Speech and Audio Processing*, 5(2):120–130, March 1997.
- [17] S. Morlat. Linphone, an open-source SIP video phone for Linux and Windows. <http://www.linphone.org/>.
- [18] Y. K. Muthusamy, E. Barnard, and R. A. Cole. Reviewing automatic language identification. *IEEE Signal Processing Magazine*, 11(4):33–41, October 1994.
- [19] J. Navrátil. Spoken language recognition—a step toward multilinguality in speechprocessing. *IEEE Transactions on Speech and Audio Processing*, 9(6):678–685, September 2001.
- [20] E. Paksoy, A. McCree, and V. Viswanathan. A variable rate multimodal speech coder with gain-matched analysis-by-synthesis. In *Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 751–754, April 1997.
- [21] N. Provos. Voice over misconfigured internet telephones. <http://vomit.xtdnet.nl>.
- [22] L. Rabiner and B. Juang. *Fundamentals of Speech Recognition*. Prentice Hall, 1993.
- [23] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session initiation protocol. RFC 3261.
- [24] M. R. Schroeder and B. S. Atal. Code-excited linear prediction(CELP): High-quality speech at very low bit rates. In *Proceedings of the 1985 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 10, pages 937–940, April 1985.
- [25] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. RFC 1889.
- [26] D. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and SSH timing attacks. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.
- [27] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu. Statistical identification of encrypted web browsing traffic. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 19–30, May 2002.
- [28] J.-M. Valin. The Speex codec manual. <http://www.speex.org/docs/manual/speex-manual.pdf>, August 2006.
- [29] J.-M. Valin and C. Montgomery. Improved noise weighting in CELP coding of speech - applying the Vorbis psychoacoustic model to Speex. In *Audio Engineering Society Convention*, May 2006. See also <http://www.speex.org>.
- [30] S. V. Vaseghi. Finite state CELP for variable rate speech coding. *IEE Proceedings I Communications, Speech and Vision*, 138(6):603–610, December 1991.
- [31] X. Wang, S. Chen, and S. Jajodia. Tracking anonymous peer-to-peer VoIP calls on the Internet. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 81–91, November 2005.
- [32] C. White, I. Shafran, and J.-L. Gauvain. Discriminative classifiers for language recognition. In *Proceedings of the 2006 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages I–213–I–216, May 2006.
- [33] E. Wong, T. Martin, T. Svendsen, and S. Sridharan. Multilingual phone clustering for recognition of spontaneous indonesian speech utilising pronunciation modelling techniques. In *Proceedings of the 8th European Conference on Speech Communication and Technology*, pages 3133–3136, September 2003.

- [34] C. V. Wright, F. Monrose, and G. M. Masson. On inferring application protocol behaviors in encrypted network traffic. *Journal of Machine Learning Research*, 7:2745–2769, December 2006. Special Topic on Machine Learning for Computer Security.
- [35] L. Zhang, T. Wang, and V. Cuperman. A CELP variable rate speech codec with low average rate. In *Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 735–738, April 1997.
- [36] Y. Zhang and V. Paxson. Detecting stepping stones. In *Proceedings of the 9<sup>th</sup> USENIX Security Symposium*, pages 171–184, August 2000.
- [37] P. Zimmermann, A. Johnston, and J. Callas. ZRTP: Extensions to RTP for Diffie-Hellman key agreement for SRTP, March 2006. IETF Internet Draft.
- [38] M. A. Zissman. Comparison of four approaches to automatic language identification of telephone speech. *IEEE Transactions on Speech and Audio Processing*, 4(1), January 1996.

## A Data Set Breakdown

The empirical analysis performed in this paper is based on one of the most widely used data sets in the language recognition community. The Oregon Graduate Institute CSLU 22 Language corpus provides speech samples from 2,066 native speakers of 21 distinct languages. Indeed, the work of Zissman [38] that we analyze in Section 4 used an earlier version of this corpus. Table 3 provides some statistics about the data set.

Language	Abbr.	Speakers	Minutes per Speaker
Arabic	AR	100	2.16
Br. Portuguese	BP	100	2.52
Cantonese	CA	93	2.63
Czech	CZ	100	2.02
English	EN	100	2.51
Farsi	FA	100	2.57
German	GE	100	2.33
Hindi	HI	100	2.74
Hungarian	HU	100	2.81
Indonesian	IN	100	2.45
Italian	IT	100	2.25
Japanese	JA	100	2.33
Korean	KO	100	2.58
Mandarin	MA	100	2.75
Polish	PO	100	2.64
Russian	RU	100	2.55
Spanish	SP	100	2.76
Swahili	SW	73	2.26
Swedish	SD	100	2.23
Tamil	TA	100	2.12
Vietnamese	VI	100	1.96

Table 3: Statistics about each language in our data set [15]. Minutes of speech is measured how many of minutes of speech we used during our tests.

# Devices That Tell On You: Privacy Trends in Consumer Ubiquitous Computing

T. Scott Saponas  
*University of Washington*

Jonathan Lester  
*University of Washington*

Carl Hartung  
*University of Washington*

Sameer Agarwal  
*University of Washington*

Tadayoshi Kohno  
*University of Washington*

## Abstract

We analyze three new consumer electronic gadgets in order to gauge the privacy and security trends in mass-market UbiComp devices. Our study of the *Slingbox Pro* uncovers a new information leakage vector for encrypted streaming multimedia. By exploiting properties of variable bitrate encoding schemes, we show that a passive adversary can determine with high probability the movie that a user is watching via her Slingbox, even when the Slingbox uses encryption. We experimentally evaluated our method against a database of over 100 hours of network traces for 26 distinct movies.

Despite an opportunity to provide significantly more location privacy than existing devices, like RFIDs, we find that an attacker can trivially exploit the *Nike+iPod Sport Kit*'s design to track users; we demonstrate this with a GoogleMaps-based distributed surveillance system. We also uncover security issues with the way Microsoft *Zunes* manage their social relationships.

We show how these products' designers could have significantly raised the bar against some of our attacks. We also use some of our attacks to motivate fundamental security and privacy challenges for future UbiComp devices.

**Keywords:** Information leakage, variable bitrate (VBR) encoding, encryption, multimedia security, privacy, location privacy, mobile social applications, UbiComp.

## 1 Introduction

As technology continues to advance, computational devices will increasingly permeate our everyday lives, placing more and more wireless computers into our environment and onto us. Many manufacturers have predicted that the increasing capabilities and decreasing costs of wireless radios will enable common electronics in future homes to be predominantly wireless, eliminating the clutter of wires common in today's homes. For exam-

ple, TVs, cable boxes, speakers, and DVD players could communicate without the proximity restrictions of wires. The changing technological landscape will also lead to new computing devices, such as personal health monitors, for us to wear on our persons as we move around our community. However, despite advances in these areas we have only just begun to see the first examples of such technologies enter the marketplace at a broad scale. While the Ubiquitous Computing (UbiComp) revolution will have many positive aspects, we must be careful to not simultaneously endanger users' privacy or security.

By studying the Sling Media Slingbox Pro, the Nike+iPod Sport Kit, and the Microsoft Zune, we provide a checkpoint of current industrial trends regarding the privacy and security of this new generation of UbiComp devices. (The Slingbox Pro is a video relay system; the Nike+iPod Sport Kit is a wireless exercise accessory for the iPod Nano; and the Zune is a portable wireless media player.) In some cases, such as our techniques for inferring information about what movie a user is watching from 10 minutes of a Slingbox Pro's encrypted transmissions, we present new directions for computer security research. For some of our other results, such as the Nike+iPod's use of a globally unique persistent identifier, the key privacy issues that we uncover are not new; but the ease with which we are able to mount our attacks is surprising. This is particularly true because we show that it would have been technically possible for the Nike+iPod designers to prevent our attacks.

In all cases, we use our results with these devices to paint a set of research challenges that future commercial UbiComp devices should address in order to provide users' with strong levels of privacy and security.

**On Our Choice of Devices.** The Slingbox Pro, the Nike+iPod Sport Kit, and the Microsoft Zune represent a cross-section of the different classes of UbiComp devices one might encounter in the future: (1) devices that permeate our environment and that stream or exchange

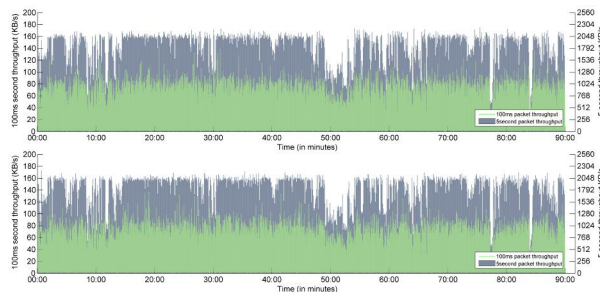


Figure 1: 5 second and 100 millisecond throughput for two traces of Ocean's Eleven played via the Slingbox and captured via a wired connection. Notice the (visual) similarity between the traces.

information; (2) devices that users have on their persons all the time; and (3) devices that promote social interactions. While there is no perfect division between these different classes of devices (e.g., devices that users have on themselves all the time may also exchange content and promote social activity), there are unique aspects to the challenges for each class of devices; we therefore consider each in turn. Specifically, (1) we use the Slingbox Pro as a vehicle to study the issues and challenges affecting next-generation wireless multimedia environments, (2) we use the Nike+iPod Sport Kit as the basis for assessing the issues and challenges with devices that we have on our persons all the time, and (3) we use the Zune as a foothold into understanding the issues and challenges with devices promoting social activity.

Below we survey our results and challenges for each of these scenarios in turn, deferring further details to the body of this paper.

## 1.1 The Sling Media Slingbox Pro

The Slingbox Pro allows users to remotely view (sling) the contents of their TV over the Internet. The makers of the Slingbox Pro are staged to introduce a new device, the wireless SlingCatcher, which will allow Slingbox users to sling video to other TVs located within the same home, thereby making it one of the first next-generation wireless video multimedia systems for the home [40]. Since the SlingCatcher will not be commercially available until later this year, we choose to study the privacy-preserving properties of a Slingbox streaming encrypted movies to a nearby computer over 802.11 wireless.

We describe in the following sections a technique for monitoring a network connection, wired or wireless, and based on the rate at which data is being sent from one device to the other, predicting the content that is being transferred. Our method consists of two parts. First, we describe a procedure for collecting throughput traces

across wired and wireless connections and combining them into a single reference trace per movie. These reference traces are collected into a database for future query use. (Figure 1 shows the raw 5 second and 100 millisecond throughput data for two wired traces of Ocean's Eleven.) Second, we describe a simple Discrete Fourier Transform based matching algorithm for querying this database and predicting the content being transmitted.

We test this algorithm on a dataset consisting of over 100 hours of network throughput data. With only 10 minutes worth of monitoring data, we are able to predict with 62% accuracy the movie that is being watched (on average over all movies); this compares favorably with the less than 4% accuracy that one would achieve by random chance. With 40 minutes worth of monitoring data, we are able to predict the movie with 77% accuracy. For certain movies we can do significantly better; for 15 out of the 26 movies, given a 40 minute trace we are able to predict the correct movie with over 98% accuracy. Given the simplicity of our algorithm, this indicates a significant amount of information leakage — a fact that is not immediately obvious to the users, who likely trust the built in encryption in the device to protect privacy.

Any transmission method whose characteristics depend on the content that is being transmitted is susceptible to the kind of attack we have described. As the world moves towards more advanced multimedia compression methods, and streaming media becomes ubiquitous, variable bitrate encoding is here to stay. Preventing information leakage in variable bitrate streams without a significant performance penalty is an interesting challenge for both the signal processing and the security communities. More broadly, a fundamental challenge that we must address is how to identify, understand, and mitigate information leakage channels in the full range of upcoming UbiComp devices.

## 1.2 The Nike+iPod Sport Kit

The Nike+iPod Sport Kit is a new wireless accessory for the iPod Nano; see Figure 2. The kit consists of two components — a wireless sensor that a user puts in one of her shoes and a receiver that she attaches to her iPod Nano. When the user walks or runs, the sensor wirelessly transmits information to the receiver; the receiver and iPod will then interpret that information and provide interactive audio feedback to the user about her workout. The Nike+iPod sensor does have an on-off button, but the online documentation suggests that most users should leave their sensors in the on position. Moreover, since the Nike+iPod online documentation encourages users to “just drop the sensor in their Nike+ shoes and forget about it [36],” the Nike+iPod Sport Kit is a prime example of the types of devices that people might even-

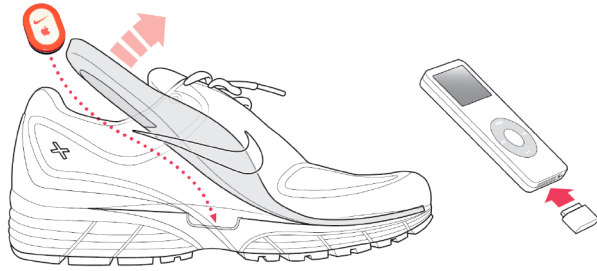


Figure 2: A Nike+iPod sensor in a Nike+ shoe and a Nike+iPod receiver connected to an iPod Nano.

tually find on themselves all the time.

One well-known potential privacy risk with having wireless devices on ourselves all the time is: if the devices use *unique identifiers* when they communicate, and if someone can intercept (sniff) those unique identifiers from the communications, then that someone might learn potentially private information about a user's presence or location. This someone might use that information in ways that are *not* in a user's best interest; e.g., a stalker might use this information to digitally track one or many people, a company might use this information for targeted advertising, and a court might examine this information when debating a contentious case. Location and tracking issues such as these are broadly discussed in the context of RFIDs [27], bluetooth devices [26, 44], and (to a lesser extent) 802.11 wireless devices [15], and there is a large body of UbiComp literature focused on privacy in location-aware systems [5, 11, 12, 19, 20, 25, 22, 29, 34]. Given this broad awareness of the potential trackability issues with wireless devices, and given media reports that the Nike+iPod Sport Kit used a proprietary wireless protocol [35] we set out to determine whether the new Nike+iPod Sport Kit proprietary system "raised the bar" against parties wishing to track users' locations.

We describe the technical process that we went through in order to discover the Nike+iPod Sport Kit protocol in Section 3. The key discovery we found is that not only does each Nike+iPod sensor have a globally unique identifier, but we can cheaply and easily detect the transmissions from the Nike+iPod shoe sensors from 10–20 meters away — an order of magnitude further than what one would expect from a wireless device that only needs to communicate from a user's shoe to the user's iPod (typically strapped around the user's arm), and also significantly further than the conventional passive RFID. The Nike+iPod sensor also broadcasts its unique identifier even when there are no iPods nearby — the user must simply be moving with a Nike+iPod sensor in her shoe. To illustrate the ease with which one could create a Nike+iPod tracking system, we devel-

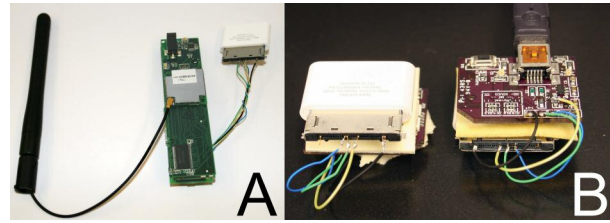


Figure 3: (a) A gumstix-based Nike+iPod surveillance device with wireless Internet capabilities. (b) Our Nike+iPod Receiver to USB adapter

oped a network of Nike+iPod surveillance devices, including a \$250 gumstix-based node. The gumstix uses an 802.11 wireless Internet connection to dynamically stream surveillance data to our back-end server, which then displays the surveillance data in a GoogleMaps application in real time.

We then describe cryptographic mechanisms that, if implemented, would have significantly improved the Nike+iPod Sport Kit's resistance to our tracking attacks, albeit with the potential drawback of additional resource consumption (e.g., battery life and communication overhead). Our basic approach is to mask the unique identifiers so that only the intended recipient can unmask them. Our solution, however, exploits the fact that the Nike+iPod Sport Kit has a very simplistic communications topology — at any given time a Nike+iPod Sport Kit sensors only needs to be able to communicate with one receiver. The challenge is therefore to lift our privacy-preserving mechanisms (or other mechanisms) to a broader context with heterogeneous devices communicating in an *ad hoc* manner.

### 1.3 The Microsoft Zune

The Microsoft Zune is a portable digital media player with one (currently) unusual feature: built in 802.11 wireless capabilities. The intended goal is to let users wirelessly share pictures and songs with other nearby Zunes — including Zunes belonging to total strangers. As such, the Zune is arguably the first major commercial device with the design goal of helping catalyze *ad hoc* social interactions in a peer-to-peer wireless environment. (Strictly speaking, we have not read the Zune design documents. Rather, we are inferring this design goal from articles in the popular press and from other publicly available information about the Zune [32].)

Unfortunately, just as it is possible for spammers to send unsolicited or inappropriate emails to users, it is possible for an attacker to beam unsolicited content to a nearby Zune. This unsolicited content may be annoying, such as advertisements or propaganda, or malicious, such as images or songs that might make the recipient

feel uncomfortable or unsafe.

Given the Zune's goal of enabling *ad hoc* interactions, the Zune cannot fall back on traditional mechanisms for preventing unsolicited content, such as buddy lists for instant messaging. Further, much of the research on social interactions for ubiquitous devices is restricted to scenarios where users have a hierarchy of social relationships (e.g., friends and non-friends) [22], which is incompatible with the assumed Zune design goals. Rather, in apparent anticipation of such unsolicited content, the Microsoft Zune allows users to "block" a particular device — a malicious individual might be able to get a user to accept an image or song once, but the recipient should be able to block the originating device from ever sending the user other content in the future. Unfortunately, we find that it is easy for an adversary to subvert this blocking mechanism, thereby allowing the adversary to repeatedly initiate content pushes to the victim until the victim walks out of range or turns off the wireless in her Zune. While we describe techniques that would address the above scenario in the particular case of the Zune, the observations we make underscore two challenges for UbiComp devices designed to enable *ad hoc* social interactions: (1) how to technically implement a blocking procedure or proactively protect against undesired content, especially among a set of heterogeneous devices, and (2) how to balance the blocking mechanisms with our desire to protect location privacy and avoid certain uses of globally unique identifiers.

## 1.4 Organization and Remarks

We respectively discuss our analyses of the Slingbox Pro, the Nike+iPod Sport Kit, and the Microsoft Zune, as well as the associated research challenges, in Sections 2, 3, and 4. We discuss related work in-line.

We stress that there is no evidence that Sling Media, Apple, Nike, or Microsoft intended for any of these devices to be used in any malicious manner. Neither Sling Media, Apple, Nike, nor Microsoft endorsed this study.

## 2 The Slingbox Pro: Information Leakage and Variable Bitrate Encoding

Although the future of home entertainment is somewhat fuzzy, many companies have predicted the future home to be a wireless one. Wireless devices tend to be easier to install (though not necessarily easier to setup), provide the user with more flexibility, allow the devices to interoperate with other technologies, and reduce clutter from wires. While it is currently easier to simply plug these devices in once and forget about them, future wireless technologies promise an ever increasing amount of bandwidth, range, and decreasing manufacturing costs,

making them more appealing and more likely to be included in future products. Consider, for example, the buzz associated with the upcoming SlingCatcher and the Apple TV; the former is expected to feature integrated wireless support; the latter currently does. In addition to the drive for devices to be connected together, wirelessly, in the home, these devices are often finding themselves networked together and connected to the Internet.

Protecting our private information becomes increasingly difficult as we begin to continually use more wireless devices. Devices in our homes could leak private information to wireless eavesdroppers or, when using home devices over the Internet, wired eavesdroppers. We have investigated one such new wireless/remote TV viewing application — the Slingbox Pro — from a privacy standpoint. In doing so we have uncovered a new information leakage vector for encrypted multimedia systems via variable bitrate encoding.

### 2.1 Slingbox Pro Description

The Slingbox Pro is a networked video streaming device built by Sling Media, Inc. It is capable of streaming video using its built in TV tuner or one of four inputs connected to DVD players, cable TV, personal video recorders, built in TV tuner, etc. and controlling these devices using an IR emitter. The device itself has no hard drive and cannot store media locally, relying on the connected devices to provide the video and audio content. Paired with player software, called SlingPlayer, the user can watch video streamed by the Slingbox Pro on their laptop, desktop, or PDA anywhere they have Internet access. To accommodate limited network connections when watching videos over a wireless network or away from home, the Slingbox Pro re-encodes the video stream using a variable bitrate encoder, likely a optimized version of Windows Media 9s VC-1 implementation [41]. The Slingbox Pro provides encryption for its data stream (regardless of any transport encryption like WPA). To avoid any problems caused by latency or network interruption the SlingPlayer will cache a buffer of several seconds worth of video. Because of this caching behavior and commonly used packet sizes for TCP packets, the data packets from the Slingbox Pro tend to always be large data packets of similar size or small (seemingly control) packets.

Sling Media recently announced a new device, the wireless SlingCatcher, which users can attach to their TVs. The SlingCatcher would allow users to wirelessly stream content from a Slingbox Pro to their TVs, thereby taking us one step further to a wireless multimedia home. Since the SlingCatcher is not yet commercially available, we choose to study the Slingbox Pro in isolation.

Index	Movie	Index	Movie
A	Bad Boys	B	Bad Boys II
C	Bourne Supremacy	D	Break-Up
E	Harry Potter 1	F	Harry Potter 3
G	Incredibles	H	Men in Black II
I	Ocean's Eleven	J	Short Circuit
K	X2	L	X-Men
M	Air Force One	N	Bourne Identity
O	Caddyshack	P	Clueless
Q	Happy Gilmore	R	Jurassic Park
S	Nightmare Before	T	Office Space
U	Red October	V	Austin Powers 1
W	Austin Powers 2	X	Bruce Almighty
Y	Hurricane	Z	Short Circuit 2

Table 1: Mapping from movie names to movie indices.

## 2.2 Experimental Setup

We ask whether Slingbox's use of encryption prevents an eavesdropper from discovering what content is being transmitted. This private information could be potentially sensitive if the content is illegal (e.g., pirated), embarrassing, or is otherwise associated with some social stigma. Toward answering this question, we conducted the following experiments.

We streamed a total of 26 movies from a Slingbox Pro to laptop and desktop Windows XP computers running the Slingmedia SlingPlayer. See Table 1. For each movie we streamed the first hour of the movie twice over a wired connection and twice over an 802.11G WPA-PSK TKIP wireless connection. Each time we used the Wireshark protocol analyzer [43] to capture all of the Slingbox encrypted packets to a file. We split each of these traces into 100-millisecond segments and calculate the data throughput for each segment. We use these 100-millisecond throughput traces as the basis for our eavesdropping analysis. See Figure 1 for two examples of these 100-millisecond traces, as well as two example 5-second throughput traces.

## 2.3 Throughput Analyses

Our eavesdropping algorithm consists of two parts. In the first part, we construct a database of reference traces. Each movie was represented by exactly one reference trace obtained by combining all the throughput traces corresponding to it. Each reference trace requires approximately 600 kilobytes of storage per hour of video. The second part of our algorithm uses this database of reference traces to match against a previously unseen trace. In the following we describe each of these two stages in detail.

**Building a Database of Reference Traces.** While it is possible to use our matching algorithm against individual raw traces, combining the raw traces for a movie into one reference trace, reduces the time complexity of the matching process and increases the statistical robustness of the matching procedure by eliminating noise and network effects peculiar to a particular trace.

For each movie, all its traces were temporally aligned with each other. This is needed because the trace capturing process was started manually and the traces could be offset in time by 0 to 20 seconds. The alignment was done by looking at the maximum of the normalized cross correlation between smoothed versions of the traces. The smoothing was performed using Savitzky-Golay filtering of degree 2 and window size 300. These filters perform smoothing while preserving high frequency content better than standard averaging filters [38]. The reference trace was obtained by averaging over the aligned raw signals.

**Matching a Query Trace to the Database.** Given a database of reference traces and a short throughput trace, we are now faced with the task of finding the best matching reference trace. This is an instance of the problem of subsequence matching in databases, which has been widely studied in both discrete and continuous domains. Our algorithm is inspired by the work of Faloutsos *et al.* [13].

The simplest approach to subsequence matching in timeseries is to calculate the Euclidean distance between the query sequence and all contiguous subsequences of the same size in the database. Due to the amount of noise present in these traces, this method does not perform well in practice. Following Faloutsos *et al.*, instead of comparing raw throughput values, we first extract noise tolerant features from the traces and then compare subsequences based on these features.

A number of feature extraction schemes have been proposed for this task in the literature, including the Discrete Fourier Transform (DFT) and the Discrete Wavelet Transform. We use the DFT in our experiments. Each point in a throughput sequence was replaced by the first  $f$  DFT coefficients of window size  $w$  centered on that point. Thus each reference trace in the database was a sequence of non-negative throughput values was replaced by a sequence of  $f$ -dimensional Fourier coefficients. The low order Fourier coefficients capture the dominant low frequency behavior in each window. We treat the higher frequency components as noise and ignore them. The same transformation is applied to the query trace. The resulting  $f$ -dimensional query trace is compared with all subsequences of the same in the database. The movie with the closest matching subsequence is declared a match. Figure 4 illustrates the database construction and matching process.

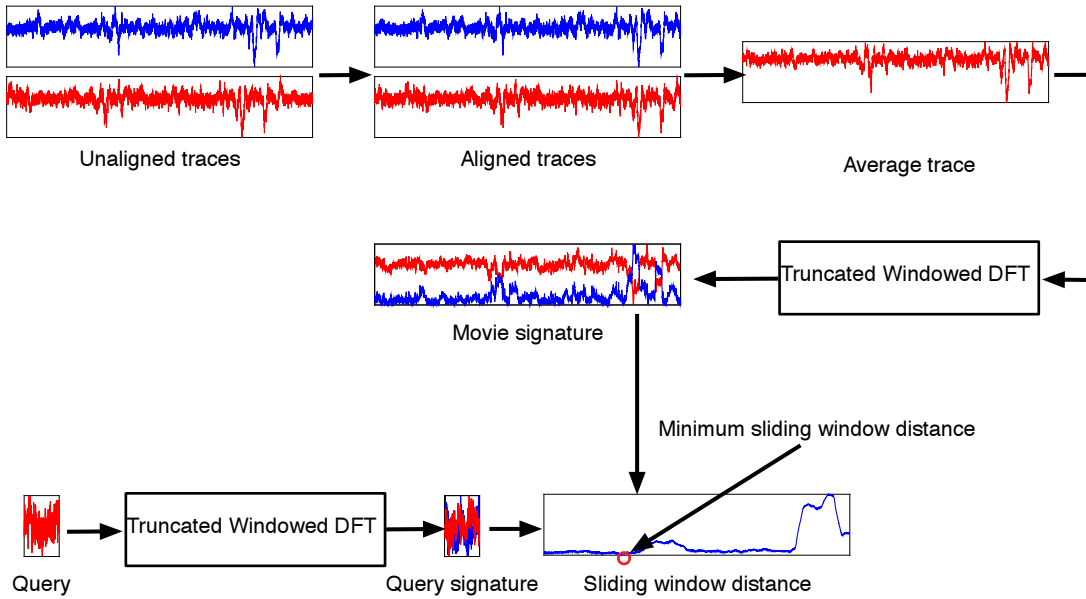


Figure 4: Database construction and query matching. The raw throughput traces corresponding to a movie are aligned and averaged to produce a single composite trace. A windowed Fourier transform is performed on the composite trace and the first  $f = 2$  coefficients are kept. A database of movie signatures is constructed in this manner. A query trace is transformed similarly into a signature, and the minimum sliding window distance between the movie signatures and the query signature is calculated. The movie with the minimum distance is declared a match.

We note that exhaustive matching of all subsequences would not be computationally feasible in a production environment with thousands of reference traces. Methods based on approximate nearest neighbor searching can be used to substantially accelerate the matching process without a significant loss in accuracy [13].

**Experiments.** The above algorithm has two parameters. The size  $w$  of the sliding window used to extract the features and the number of Fourier features  $f$ , extracted from each window. Both affect the recognition performance of the algorithm. Small values of  $w$  and  $f$  result in high noise sensitivity, and large values result in over-smoothing of the data. The other factor that affects recognition performance is the length  $l$  of the query trace. To choose a good parameter setting, we studied the behavior of the algorithm described above for varying values of  $w = [100, 300, 600]$ ,  $f = [1, 2, 4]$ . For each setting of the parameters, a random query trace of length  $l = 6000$  was extracted from one of the raw throughput traces and compared using the matching algorithm described above. This procedure was repeated 100 times for every parameter setting. The highest accuracy was obtained for  $w = 100$  and  $f = 2$ , or a sliding window of 10 seconds with two Fourier coefficients per window.

We now fix  $w = 100$  and  $f = 2$  parameters, vary  $l = [6000, 12000, 18000, 24000]$  (10, 20, 30, and 40 minutes), and estimate the prediction accuracy of the

eavesdropping algorithm. This is done by choosing one throughput trace at a time, constructing the reference trace database using the rest of the throughput traces and then counting how many times random subsequences from the chosen trace result in an incorrect prediction. The average number of incorrect matches over all traces is the leave one out error [18]. In our experiment, 50 random subsequences were chosen from each trace. Sometimes a good shortlist of possible matches is also useful, where the list can be further trimmed with side information, for example, the cable schedule for the area. To account for this possibility, not only do we count the number of times we get the best match right, we also count for varying values of  $k = 1, \dots, 5$ , when the algorithm correctly ranks the movie amongst the top  $k$  matches.

Table 2 reports the overall accuracy (1-error) of the algorithm, where the accuracy (true positive rate) was computed over all 26 movies. (We define the true positive rate of a movie  $M$  as the rate at which a random query trace for movie  $M$  is correctly identified as movie  $M$ ; we define the false positive rate of a movie  $M$  as the rate at which a random query trace for a movie  $M' \neq M$  is incorrectly identified as movie  $M$ .)

For 10- and 40-minute queries, the overall accuracy rates are respectively 62% and 77%. Table 3 and Figures 5 and 6 show that the accuracy rate for individual movies can be significantly higher. From Table 3, 15 of our 26 movies had  $\geq 98\%$  true positive rates for 40-

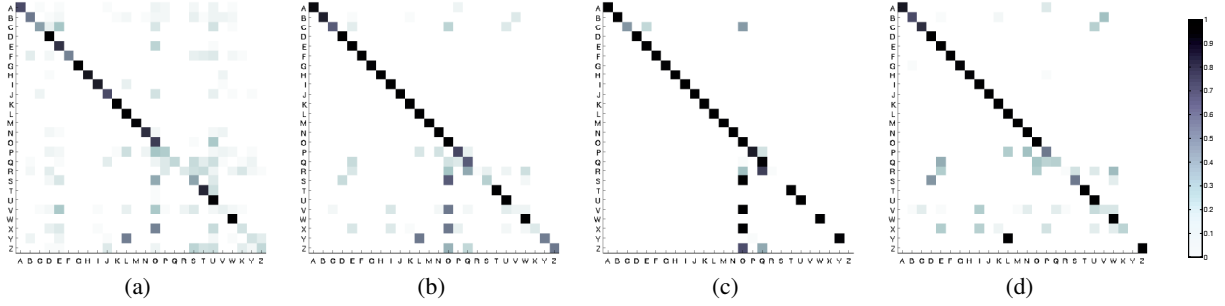


Figure 5: Confusion matrices for: (a) 10 minute probes from both wired and wireless traces; (b) 40 minute probes from both wired and wireless traces; (c) 40 minute probes from wired traces; (d) 40 minute probes from wireless traces. The color scale is on the right; black corresponds to 1.0 and white corresponds to 0.0.

	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
10 mins	0.62	0.66	0.69	0.71	0.73
20 mins	0.71	0.75	0.78	0.80	0.82
30 mins	0.74	0.79	0.81	0.84	0.85
40 mins	0.77	0.81	0.84	0.86	0.89
chance	0.04	0.08	0.12	0.15	0.19

Table 2: Overall accuracy of the eavesdropping algorithm. The rows correspond to 10, 20, 30, and 40 minute query traces, and the columns report the success with which the algorithm correctly placed the movie in the top  $k$  matches. The bottom row correspond to the probability of a match by random chance.

minute traces with  $k = 1$ , and 22 of our 26 movies had  $\leq 1\%$  false positive rates for our 40-minute traces with  $k = 1$ .

Figures 5 (a) and (b) show the confusion matrices for 10 and 40 minute query traces with  $k = 1$ . The shade of the cell in row  $i$ , column  $j$  denotes the rate at which the  $i$ -th movie is identified as the  $j$ -th movie; the cells on the diagonal correspond to correct identifications. Contrasting Figures 5 (a) and (b) visually show the increase in accuracy as the length of the query trace increases. Our wireless traces have a higher level of noise as compared to our wired traces. Figures 5 (c) and (d) therefore show the confusion matrices for when the query is restricted to (c) wired and (d) wireless traces. Note that a few movies were misidentified as Caddyshack, as represented by the vertical band most visible in Figure 5 (c); this is likely due to the fact that the bitrate for Caddyshack was fairly constant and the misidentified movies had significant noise (e.g., the wireless traces for Austin Powers 1 had significant noise, which influenced the composite reference trace and therefore the ability of the Austin Power query trace to match to the reference trace).

Movie Index	True positives $n$ minute probes		False positives $n$ minute probes	
	$n = 10$	$n = 40$	$n = 10$	$n = 40$
A	0.67	0.95	0.00	0.00
B	0.51	0.86	0.01	0.00
C	0.38	0.60	0.01	0.00
D	1.00	1.00	0.02	0.01
E	0.78	1.00	0.04	0.02
F	0.47	0.99	0.00	0.00
G	0.99	0.98	0.00	0.00
H	0.90	0.99	0.00	0.00
I	0.87	1.00	0.00	0.01
J	0.66	1.00	0.01	0.00
K	0.99	0.99	0.00	0.00
L	0.99	1.00	0.03	0.02
M	0.98	0.99	0.00	0.00
N	0.79	1.00	0.00	0.01
O	0.72	1.00	0.08	0.10
P	0.22	0.68	0.01	0.01
Q	0.18	0.59	0.00	0.02
R	0.10	0.02	0.00	0.00
S	0.37	0.20	0.03	0.00
T	0.83	1.00	0.02	0.00
U	0.97	1.00	0.06	0.01
V	0.07	0.06	0.01	0.01
W	0.99	1.00	0.01	0.01
X	0.14	0.10	0.01	0.00
Y	0.12	0.49	0.01	0.00
Z	0.18	0.50	0.01	0.00

Table 3: True and false positive rates for 10 and 40 minute probes of both wired and wireless traces. The true positive rate of a movie  $M$  is the rate at which an  $n$ -minute query of that movie is correctly identified as movie  $M$ . The false positive rate of a movie  $M$  is the rate at which an  $n$ -minute query of some other movie  $M' \neq M$  is incorrectly identified as  $M$ .

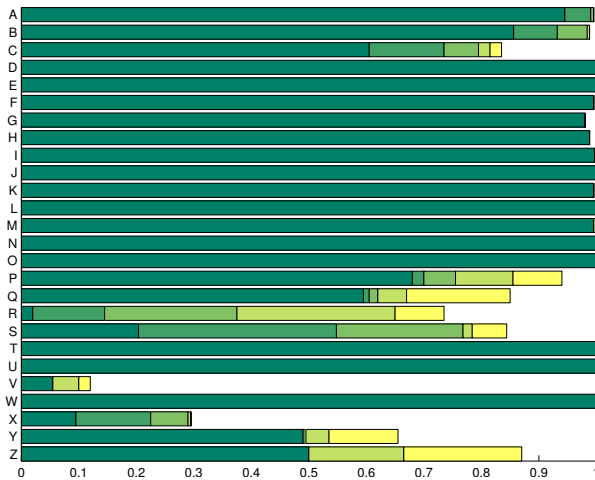


Figure 6: Accuracy per movie for 40 minute query traces;  $k = 1$  through  $k = 5$ .

## 2.4 Limitations, Implications, and Challenges

While our experiments were conducted in a laboratory setting, they do reflect some possible configurations that one might encounter in a future home equipped with many wireless multimedia devices. The implications of our results are, therefore, that an adversary in close proximity to a users' home might be able to infer information about what videos a user is watching. This adversary might be a nosy neighbor. Or the adversary might be someone sitting outside in a van, looking to collect forensics evidence about those viewing "illegal" (e.g., censored or pirated) content. Moreover, a content producer (such as the creator of a movie) could intentionally construct its movies to have stronger, more distinctive fingerprints. This situation would seem to violate the user's perception of privacy within their own home, especially given the Slingbox Pro's use of encryption.

More broadly, our Slingbox results provide further evidence that encryption alone cannot fully conceal the contents of encrypted data. Other results show that one can infer the origins of encrypted web traffic or infer application protocol behaviors from encrypted data [30, 45]. Concurrent with this work, Wright *et al.* show how variable bitrate encodings can reveal the language spoken through an encrypted VoIP connection [46]. Protecting against such information leakage vectors for all possible applications seems to be a fundamental challenge. Indeed, it may be difficult to simultaneously preserve desirable properties like low-latency and low bandwidth consumption while also allowing for applications with bursty or otherwise data-dependent communication properties. As a concrete example, while it may be possi-

ble to significantly raise the bar against information leakage through the Slingbox by having the Slingbox push data at a constant rate while a user is watching a movie, a passive eavesdropper may still be able to learn when a user watches movies, and for how long. The challenge, therefore, is to first determine the possible information leakage vectors, understand their implications, and develop technical means for mitigating them.

## 3 The Nike+iPod Sport Kit: Devices that Reveal Your Presence

The Nike+iPod Sport Kit foreshadows the types of application-specific UbiComp devices that we might soon find ourselves wearing as part of our daily routine. Indeed, based on publicly available information about the intended usage of the Nike+iPod Sport Kit, as well as our own personal observations, we expect that many Nike+iPod users will always leave their Nike+iPod sensors turned on and in their shoes.

We describe here the steps we took to discover the Nike+iPod protocol; our goal was to assess whether the Nike+iPod Sport Kit provides protection mechanisms against an adversary who wishes to track users' locations. Having uncovered no such protection mechanisms, we then describe our subsequent steps to gauge how easy and cheap it might be for an adversary to implement our attacks. Finally we consider fixes to the Nike+iPod protocol as well as some broader research challenges that our results raise.

### 3.1 Nike+iPod Description

The Nike+iPod Sport Kit allows runners and walkers to hear real time workout progress reports on their iPod Nanos. A typical user would purchase an iPod Nano, a Nike+iPod Sport Kit, and either a pair of Nike+ shoes or a special pouch to attach to non-Nike+ shoes. The kit consists of a receiver and a sensor. Users place the sensor in their left Nike+ shoe and attach the receiver to their iPod Nano as shown in Figure 2. The sensor is a 3.5cm x 2.5cm x 0.75cm plastic encased device, and the receiver is a 2.5cm x 2cm x 0.5cm plastic encased device. When a person runs or walks the sensor begins to broadcasts sensor data via a radio transmitter whether or not an iPod Nano is present. When the person stops running or walking for ten seconds, the sensor goes to sleep. When the iPod Nano is in *workout mode* and the receiver's radio receives sensor data from the sensor, the receiver will relay (a function of) that data to the iPod Nano, which will then give audio feedback (via the iPod headphones) to the person about his or her workout. As of September 2006, Apple has sold more than 450,000 of the \$29 (USD) Nike+iPod Sport Kits [1].

### 3.2 Discovering the Nike+iPod Protocol

**Initial Analysis.** The first step was to learn how the Nike+iPod sensor communicates with the receiver. According to the Nike+iPod documentation, a sensor and receiver need to be *linked* together before use; this linking process involves user participation. Once linked, the receiver will only report data from that specific sensor, eliminating the readings from other nearby sensors. The receiver can also remember the last sensor to which it was linked so that users do not need to perform the linking step every time they turn on their iPods. The receiver can also later be linked to a different sensor (for a replacement sensor or different user), but under the standard user interface the receiver can only be linked to one sensor at any given time.

We observed, however, that a single sensor could be linked to two receivers simultaneously, meaning that two people could use their iPod Nanos and the standard user interface to read the data from a single Nike+iPod sensor at the same time. Further investigation revealed that the sensor was a transmitter only, meaning that it was incapable of knowing what iPod or receiver it was associated with. This observation provides the underlying foundation for our results since it concretely shows that a Nike+iPod Sport Kit does not enforce a strong, exclusive, one-to-one binding between a sensor and a receiver. Having made this observation, we then commenced to uncover more details about the Nike+iPod protocol.

**The Hardware, Serial Communications, and Unique Identifiers.** The Nike+iPod Sport Kit receiver communicates with the iPod Nano through the standard iPod connector. Examining which pins are present on the receiver's connector and comparing those pins with online third-party pin documentation [24], we determined that communication was most likely being done over a serial connection.

Opening the white plastic case of the receiver reveals a component board and the pin connections to the iPod connector. There are ten pins in use; three of these pins are used in serial communication: ground, iPod transmit, and iPod receive. We verified that digital data was being sent across this serial connection using an oscilloscope and soldered wires connecting them to the serial port of our computer. With the receiver connected to the iPod we turned on the iPod and observed data sent in both directions over the serial connection.

As noted above, before the receiver can be used with a new sensor, the sensor must be *linked* with the receiver. This is initiated by the user through menus in the iPod interface. The user is asked to walk around so that the sensor can be detected by the receiver. When the link process is started, the iPod sends some data to the receiver. Then, the receiver begins sending data back to

the iPod until the new sensor is discovered and linked by the receiver. Finally, the iPod sends some more data back to the receiver.

After collecting and comparing several traces of the link process with several different sensors we noticed that linking seemed to complete when the third occurrence of a certain packet came from the receiver. These packets' payload started with the same four bytes; however, the next four bytes were different depending on which sensor we used. In all our experiments these four bytes appear to be consistent and unique for a single sensor, and therefore we refer to these four bytes as the sensor's *unique identifier* or *UID*. As further corroboration for the uniqueness of these UIDs, we find that we can use the iPod Nano as an oracle for translating between the UIDs and the Nike+iPod sensor's serial number as it appears on the back of the sensor; we omit details but instead refer the reader to Figure 7 for a sketch of how one might use an iPod Nano as a UID to serial number oracle. As suggested above, the Nike+iPod Sport Kit appears to use these UIDs for addressing purposes — after linking, a receiver will only report packets containing the specified UID.

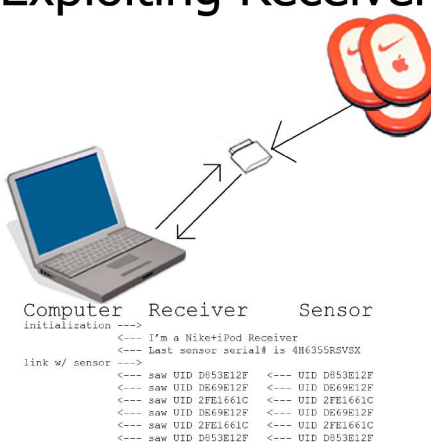
**Automatically Discovering UIDs.** Our next step was to use the Nike+iPod receiver to listen for sensor UIDs in an automated fashion *without* the iPod Nano. To do this we modified an iPod female connector by soldering wires from the serial pins on the iPod connector to our adapter, adjusted the voltage accordingly, and attached 3.3V power to the power pin. We then plugged an unmodified Nike+iPod receiver into our female connector and replayed the data that we saw coming from the iPod when the iPod is turned on and then when the iPod enters link mode. This process caused the receiver to start sending packets over the serial connection to our computer with the identifiers of the broadcasting sensors in range. However, because our computer never responds to the receiver's packets, the link process never ends and the receiver continues to send to our computer the identifiers of transmitting sensors until power is removed.

**Implications.** Our observations here immediately imply that the Nike+iPod Sport Kit may leak private information about a user's location. Namely, as is well known in the context of other devices (like RFIDs and discoverable bluetooth devices [26, 27, 44]), if a wireless device broadcasts a persistent globally unique identifier, an attacker with multiple wireless sniffers can correlate the location of that device (and by inference the user) across different physical spaces and over time.

## Listening In



## Exploiting Receiver



## Translating UID

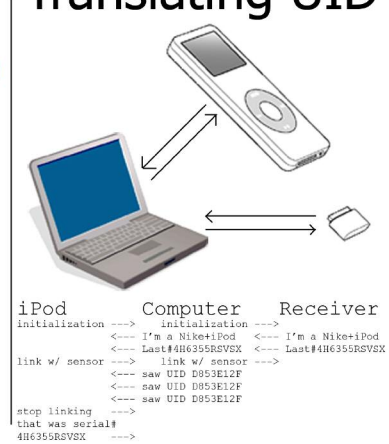


Figure 7: The figure on the left shows our approach for passively monitoring the serial communications between an iPod and the Nike+iPod receiver; the communications between the iPod and the receiver are over a physical, serial connection, and the communication from the sensor to the receiver is via a radio. The figure in the middle shows our approach for directly controlling a Nike+iPod receiver from a computer; the communication from the computer to the Nike+iPod receiver is over a physical serial connection. The figure on the right shows our approach for translating between a sensor's UID and the sensor's serial number.

### 3.3 Measurements

To understand the implications of our observations in Section 3.2, we must understand the following properties of a Nike+iPod sensor: when it transmits; how often it transmits; the range at which the receiver hears the sensor's UID; and the collision behavior of multiple sensors. We have already partially addressed some of these properties, but elaborate on our observations here.

When the sensor is still, it is “sleeping” to save battery. When one begins to walk or run with the sensor in their shoe, the sensor begins transmitting. It is also possible to wake up the sensor without putting it in a shoe. For example, shaking the sensor while still in the sealed package from the store will cause it to transmit its UID. Sensors can also be awakened by tapping them against a hard surface or shaking them sharply. Similarly, if a sensor is in the pocket of one's pants, backpack, or purse, it will occasionally wake up and start transmitting. Once walking, running, or shaking ceases, the sensor goes to sleep after approximately ten seconds.

While the sensor is awake and nearby we observed that it transmits one packet every second (containing the UID). When the sensor is more distant or around a corner the receiver heard packets intermittently, but still on second intervals. When multiple sensors are awake near one another some packets get corrupted (their checksums do not match). As the number of awake sensors increase so does the number of corrupt packets. However, our tests with seven sensors indicated the receiver still hears

every sensor UID at least once in a ten second window. During our experiments with the Nike+iPod sensors we observed approximately a 10 meter range indoors and a 10–20 meter range outdoors. Sensors are also detectable while moving quickly. Running by a receiver at approximately 10 MPH, the sensor is reliably received. Driving by someone walking with a sensor in their shoe, the sensor can be reliably detected at 30 MPH. We have not tested faster speeds.

### 3.4 Instrumenting Attacks

Section 3.2 shows that it is possible for an adversary to extract a Nike+iPod sensor's UIDs from sniffed radio transmissions, and Section 3.3 qualifies the circumstances under which the receiver might be able to sniff those transmissions. These results already enable us to conclude that, despite broad awareness about the trackability concerns with unique identifiers in other technologies (e.g., RFIDs, discoverable bluetooth), new commercial products are still entering the market without any strong protection mechanisms for ensuring users' location privacy.

We now seek to explore just how easy — in terms of cost and technical sophistication — it might be for an adversary to exploit the Nike+iPod Sport Kit's lack of location privacy protection and, at the same time, to explore the types of applications that an adversary might build. For example, one application that we built is a GoogleMaps-based system that pools data from multiple

Nike+iPod sniffs and displays the resulting tracking information on a map in real-time. When assessing the ease with which an attacker might be able to implement a Nike+iPod-based surveillance system, it is worth noting that the attacker may not need to write source code him or herself, but may instead download the necessary software from somewhere on the Internet. We built the following components and systems:

- **Receiver to USB Adaptor.** We created a compact USB receiver module for connecting the Nike+iPod receiver to a computer via USB. Our module does not require any modification to the Nike+iPod receiver; see Figure 3b, and consists of a female iPod connector [23] and a serial-to-USB board utilizing the FTDI FT232C chipset [14]. We connected the serial pins and power pins of the iPod connector to the appropriate pins of the FT232C board. When this module is connected to a computer, the receiver is then powered and a USB serial port is made available for our software to communicate with the receiver. With the receiver attached, this package is approximately 3cm x 3cm x 2cm.

We also created a windows serial communications tool for interfacing with the Nike+iPod Receiver using our adaptor. Our tool can detect the UIDs of nearby Nike+iPod sensors and transmit those UID readings, a timestamp, and latitude and longitude information to a back-end SQL server for post-processing; the latitude and longitude are currently set manually. Optionally, when a sensor is detected, this application can take photographs with a USB camera and upload those photographs to the SQL server along with the UID information. This application can also SMS or email sensor information to pre-specified phone numbers or email addresses.

- **Gumstixs.** We also implemented a cheap Nike+iPod surveillance device using the Linux-based gumstix computers. This module consists of an unmodified \$29 Nike+iPod receiver, a \$109 gumstix connex 200xm motherboard, a \$79 wifistix, a \$27.50 gumstix breakout board, and a \$2.95 female iPod connector. The Nike+iPod receiver is connected directly to the gumstix's serial port, thereby eliminating the need for our serial-to-USB adaptor. The assembled package is 8cm x 2.1cm x 1.3cm and weighs 1.1 ounces; see Figure 3a.

Our gumstix-based module runs a 280 line C program that communicates with the Nike+iPod receiver over a serial port and that uses the wifistix 802.11 wireless module to wirelessly transmit real-time surveillance data to a centralized back-end server. The real-time reporting capability allows the gumstix module to be part of a larger real-time surveil-

lance system. If an adversary does not need this real-time capability, then the adversary can reduce the cost of this module by omitting the wifistix.

- **A Distributed Surveillance System.** To illustrate the power of aggregating sensor information from multiple physical locations, we created a GoogleMaps-based web application. Our web application uses and displays the sensor event data uploaded to a central SQL server from multiple data sources. The data sources may be our serial communication tool or our gumstix application.

In real-time mode, sensors' UIDs are overlaid on a GoogleMaps map at the location the sensor is seen. When the sensor is no longer present at that location, the UID disappears. Optionally, digital pictures taken by a laptop when the sensor is first seen can be overlaid instead of the UID. In history mode, the web application allows the user to select a timespan and show all sensors recorded in that timespan. For example, one could select the timespan between noon and 6pm on a given day; all sensors seen that afternoon will be overlaid on the map at the appropriate location.

This application would allow many individuals to track people of interest. An attacker might also use this tool to establish patterns of presence. If many attackers with receivers cooperated, this software and website would allow the tracking and correlation of many people with Nike+iPod sensors. Among the related research, demonstration, and commercial bluetooth- and 802.11 wireless-based tracking systems (e.g., [6, 8, 10, 17, 31, 37, 39]), we are unaware of any other location-based surveillance system that goes as far as plotting subjects' locations on a map in real-time.

We also developed two other surveillance devices — one which uses a third-generation iPod and iPod Linux to detect nearby Nike+iPod sensors, and the other of which uses a second-generation Intel Mote (iMote2) to detect nearby Nike+iPod sensors and beams the recorded information to a paired Microsoft SPOT watch via bluetooth. For brevity, and since the above applications provide a survey of the applications that we developed, we omit discussion of our iPod Linux- and iMote2-based applications here.

### 3.5 Privacy-Preserving Alternatives

Our results show that, despite public awareness of the importance of location privacy and untrackability, major new products are still being introduced without strong privacy guards. We consider this situation unfortunate since in many cases it is technically possible to signifi-

cantly improve consumer privacy.

**Exploiting (Largely) Static Associations.** Consider the typical usage scenario for the Nike+iPod Sport Kit. In the common case, we expect that once a user purchases a Nike+iPod Sport Kit, he or she will rarely use the sensor from that kit with the receiver from a different kit. This means that the sensor and the receiver could have been pre-programmed at the factory with a shared secret cryptographic key. By having the sensor encrypt each broadcast message with this shared key, the Nike+iPod designers could have addressed most of our privacy concerns about the Nike+iPod application protocol; there may still be information leakage through the underlying radio hardware, which would have to be dealt with separately. If the manufacturer decides a sensor from one kit should be used with the receiver from a separate kit, then several options still remain. For example, under the assumption that one will only rarely want to use a sensor from one kit with a receiver from another, the cryptographic key could be written on the backs of the sensors, and a user could manually enter that key into their iPods or computers before using that new sensor. Alternately, the sensor could have a special button on it that, when pressed, causes the sensor to actually broadcasts a cryptographic key for some short duration of time.

**Un-Sniffable Unique Identifiers.** Assume now that both the sensor and the receiver in a Nike+iPod Sport Kit are preprogrammed with the same shared 128-bit cryptographic key  $K$ . One design approach would be for the sensor to pre-generate a new pseudorandom 128-bit value  $X$  during the one-second idle time between broadcasts. Although the sensor could generate  $X$  using physical processes, we suggest generating  $X$  by using AES in CTR mode with a second, non-shared 128-bit AES key  $K'$ . Also during this one-second idle time between broadcast, the sensor could pre-generate a keystream  $S$  using AES in CTR mode, this time with the initial counter  $X$  and the shared key  $K$ . Finally, when the sensor wishes to send a message  $M$  to the corresponding receiver, the sensor would actually send the pair  $(X, M \oplus S)$ , where “ $\oplus$ ” denotes the exclusive-or operation. Upon receiving a message  $(X, Y)$ , the receiver would re-generate  $S$  from  $X$  and the shared key  $K$ , recover  $M$  as  $Y \oplus S$ , and then accept  $M$  as coming from the paired sensor if  $M$  contains the desired UID. This construction shares commonality with the randomized hash lock protocol for anonymous authorization [42] in which an RFID tag reader must try all tag keys in order to determine the identity of an RFID tag; in our case a receiver must attempt to decrypt all received messages, even when the messages are intended for other receivers. While it is rather straightforward to argue that this construction provides privacy at the application level against

passive adversaries (by leveraging Bellare *et al.*'s [4] provable security results for CTR mode encryption), we do acknowledge that this construction may not fully provide all desired target security properties against active adversaries. Furthermore, we acknowledge that there are ways of optimizing the approach outlined above, and that the above approach may affect the battery life, manufacturing costs, and usability of the Nike+iPod Sport Kit.

**Use an On-Off Switch.** One natural question to ask is whether a sufficient privacy-protection mechanism might simply be to place on-off switches directly on all mobile personal devices, like the Nike+iPod Sport Kit sensors. Unfortunately, this approach by itself will not protect consumers' privacy while the devices are in operation. Additionally, we believe that it is unrealistic to assume that most users will actually turn their devices off when not in use, especially as the number of such personal devices increases over time.

### 3.6 Challenges

While the above discussion clearly shows that it is possible to significantly improve upon the privacy properties of the current Nike+iPod Sport Kits, from a broader perspective the solutions advocated above are somewhat unsatisfying. For example, how does one generalize the above recommendations (or derive new recommendations) for wireless devices that do not have largely static pairings, such as commercial 802.11 wireless hot spots or the dynamic peer-to-peer pairings of the Zune, where one may wish to allow for *ad hoc* network formations but still restrict access to only authorized devices? And how does one reduce the extra costs (e.g., battery lifetime, packet size, the need to decrypt packets intended to other parties), to environments that cannot afford the extra resource requirements? If we wish to provide a strong level of location privacy for future UbiComp devices, we need to develop mechanisms for handling such broad classes of situations.

The challenge, therefore, is to provide anonymous communications for wireless devices in more diverse and potentially *ad hoc* environments. This challenge is not unique to us — indeed, others have also considered this problem in other restricted contexts [16, 21, 33, 42, 28, 33, 44] — but bears repeating given the potential complexities; e.g., while we have focused this discussion on unique identifiers, which by themselves are not trivial to address, application characteristics and other side channel information, which can survive encryption [30, 45], might facilitate the tracking and identification of individuals.

## 4 Zunes: Challenges with Managing Ad Hoc Mobile Social Interactions

The Microsoft Zune portable media player is one of the first portable media devices to include wireless capability for the purpose of sharing media. Zune owners can enter a coffee shop, turn on their Zune, and discover nearby Zunes. Once a nearby Zune is discovered, users can send music or photos to the nearby Zune. Discovery and sharing are meant to facilitate social interaction; hence the Zune slogan: “Welcome to the Social.” Like the Nike+iPod Sport Kit and SlingBox, the Zune represents a gadget pioneering a new application space and represents a central example of our third class of UbiComp devices geared toward catalyzing new social interactions. However, we demonstrate that there are challenges with protecting users’ privacy and safety while simultaneously providing ad hoc communications with strangers.

### 4.1 Zune Description

We focus this description on how the Zune media player allows users to control their social interactions. Consider a scenario consisting of two users, Alice and Bob, and assume that Alice and Bob respectively name their Zunes AliceZune and BobZune; Alice and Bob choose these names when they configure their Zune. If Bob wishes to utilize the Zune social system, to see who’s around, he would first use the Zune interface navigate to the “community – nearby devices” menu. He will then see the names of all discoverable nearby Zunes and, depending on the options chosen by the owners of the other Zunes, the names of the songs that his neighbors are listening to or their state (online/busy). If Bob wishes to share a song or picture with his neighbors, he must first select the song or picture and then select the “send” option. The Zune will then show Bob the names of nearby Zunes, and Bob can then send the song or picture to a neighbor of his choosing, in this case AliceZune. The interface on Alice’s Zune asks whether Alice wishes to accept a song from BobZune; no additional information about the song or picture is included in the prompt. Alice has two choices: to accept the content or to not accept the content. If Alice accepts the song and later decides that she would like to prevent Bob from ever sending her a song in the future, she can navigate to her Zune’s “community – nearby devices” menu, select BobZune, and then select the “block” option.

### 4.2 Circumventing the Zune Blocking Mechanism

Microsoft appears to envision a world where Zune owners wish to receive interesting content from people they have never met before. Of course, these users also wish to avoid being bothered by people or companies that send inappropriate or annoying content, hence the Zune’s blocking feature. Such a situation is not purely hypothetical; indeed, there has recently been media reports about advertisers beaming unsolicited content to users with discoverable Bluetooth devices [7].

Unfortunately, we find that a malicious adversary could circumvent the Zune blocking feature, and we have verified this in practice. The critical issue revolves around how blocking is actually implemented on the Zunes. When Bob sends a song or image to Alice, Alice is only given the option of accepting or denying the song or image; she is not given the option of blocking the sender. Then, after playing the song or viewing the image, if Alice wishes to block Bob’s Zune in the future, she must navigate to the “community – nearby devices” menu and actively choose to block BobZune.

The crux of the problem is that Alice will not be able to block Bob’s Zune if BobZune is no longer nearby or discoverable.

**Disappearing attack Zune.** A simple method to circumvent the Zune block feature is, after beaming an inappropriate image, to turn the wireless on the originating Zune off. Since Alice may remember the name of Bob’s Zune, and thereby simply deny messages from BobZune in the future, Bob can change the name of his Zune before trying to beam Alice additional content. Also, before beaming Alice the inappropriate content in the first place, Bob could scan his nearby community, find a nearby Zune named CharlieZune, and then name his Zune CharlieZune. If Bob sends inappropriate content to Alice and then turns off his wireless, he might trick Alice into blocking the real CharlieZune.

**Fake MAC addresses.** Upon further investigation, we find that the Zune neighbor discovery process and blocking mechanism is based on 802.11 probe-responses and MAC addresses. Bob could therefore use a Linux laptop to fool Alice into thinking that she has blocked BobZune when in fact she has not; unlike the observation in the previous paragraph, our attack here works even when there are no other nearby Zunes.

Building on the scenario above, where Bob sends inappropriate content to Alice, disables his Zune’s wireless, and changes his Zune’s name. Suppose Alice does not like the content she received from Bob and navigates to the nearby list on her Zune. Bob can use his laptop to send out Zune 802.11 probe-responses with the same

name that his Zune was using but with a different MAC address. Alice will then see the previous name of Bob's Zune in her nearby list and select the block command. It will now appear to Alice that she has blocked Bob's Zune. Conversely, what has actually occurred is Alice has blocked a different MAC address. The next time Bob enables his Zune's wireless and attempts to send inappropriate content to Alice, it will appear to Alice that Bob is sending content from a third BobZune that Alice has never seen before. We have implemented a C application for Linux that uses the MadWiFi drivers and an Atheros Chipset-based wireless card to listen to 802.11 probe-requests from Zunes and send a Zune probe-response with whatever name and MAC address the user desires.

**Post-blocking privacy.** Lastly, even when the blocking mechanism is used successfully, it only stops Alice from receiving new content pushes from BobZune; the malicious user, Bob, can still detect Alice's presence unless she turns off her Zune's wireless capability all together; this has the negative side effect of preventing Alice from sharing any media at all if she doesn't want to be detectable by Bob.

### 4.3 Improving User Control

Perhaps the most natural method for protecting against such unsolicited content is to adopt what is now common practice in other social applications, such as instant messaging: create a "buddy list" and only accept connections from known buddies. One might populate the buddy list using some interactions that require two Zunes to be in close proximity [2]. Such a buddy list is, however, in direct conflict with the Zune's intended goal of initiating *ad hoc* interactions with total strangers.

Therefore, the goal is to improve the resistance of the Zune blocking mechanisms to attacks like those we present above. One simple solution to Bob's blocking circumvention is to record which Zune sent the specific media and allow the user to block the *sender* of media even if they are not currently nearby and active. We note, however, that there are some subtleties that one must consider. For example, since the Zune blocking mechanism described above seems to be based on the Zune's MAC address (recall that our C program in Section 4.2 created 802.11 probe-responses with forged MAC addresses to trick the Zune blocking feature) Bob might still be able to circumvent this improved blocking mechanism by mounting a MAC-rewriting man-in-the-middle attack between his Zune and Alice's. Since the Zune's communicate using encryption, MAC rewriting of this form will not, however, be successful if the Zunes' MAC addresses are used as input to the encryption key derivation process. We have currently not successfully determined whether or not this is actually the case, but argue

below that the use of MAC addresses for this purpose is fundamentally problematic if one also wishes to protect information about a user's presence to outsiders (recall Section 3).

### 4.4 Challenges

While there has been significant research on providing control over private information in social networks in ubiquitous social applications, much of the work focuses on situations with hierarchical or other complex relationships, such as boss/spouse/friend or buddies/non-buddies [22]. While there is still much work to be done in this space, the Zunes suggest another scenario in which a key target application is to share content with strangers. Blocking individuals in such a scenario can be very challenging when users have complete control over the information that their devices present to others.

When all the devices are homogeneous and incorporate a secure hardware module, one possibility is to let that secure hardware control what information is shared with the user and other devices, and to ensure that some information (such as a unique identifier) is not mutable by the user. The secure hardware might then use this non-mutable information to control blocking. Coupled with the discussion in Section 3, one must ensure that these unique identifiers do not reveal private information about a user's presence. For example, this unique identifier should not be an 802.11 MAC address, which the Zunes currently appear to use for blocking purposes. While there might be approaches for addressing this problem in the case of homogeneous devices with secure hardware from the same manufacturer (e.g., restricted behavior on the secure hardware and symmetric key agreement using the exchange of anonymous public keys [3] signed using a group signature scheme [9]), solving this problem in the case of a heterogeneous environment appears to be a challenge.

## 5 Conclusions

We technically explore privacy and security properties of several commercial UbiComp products. We find that despite research and public awareness, these products do not provide strong levels of privacy protection and do not put the user in control of their private information.

Our analysis of the encrypted SlingBox stream suggests that transmission characteristics from variable data rate encoding can cause information leakage even when such a stream is encrypted. This puts users privacy at risk because one might assume encryption is enough to thwart an eavesdropper from learning what media one is watching. Our first attempt at recognizing movies via their variable throughput in a 26 movie database yielded

an overall accuracy of approximately 62% for the best match and 73% for ranking in the top 5 matches when a 10 minute query trace was used, and 77% and 89% respectively when a 40 minute query trace was used; which compared with the 4% and 21% respectively that one can expect with random guessing, shows much information leakage. For certain movies our accuracy rates are significantly higher; for example, for 15 out of our 26 movies, a 40-minute query trace will match with the correct movie over 98% of the time. When a variable data rate encoding is used, a content provider could potentially increase this accuracy by using a throughput-based watermarking scheme.

Persistent identifiers in the Nike+iPod Sport Kit and Zune potentially reveal presence and, in the Nike+iPod case, we demonstrate how a tracking system can be built using the Nike+iPod Sport Kit sensors and receivers. We argue that these persistent identifiers should not be used in future devices and should instead be replaced with other privacy preserving mechanisms.

Finally, our evaluation of the Zune blocking scheme shows that an interface design choice coupled with a technology choice can take control away from the consumer and put it in the hands of malicious users. Together, the results from this paper demonstrate with new classes of devices come new privacy and security challenges; privacy must be designed in at all levels of the protocol stack.

## Acknowledgments

We thank Yaw Anokwa, Kate Everitt, Kevin Fu, J. Alex Halderman, Karl Koscher, Ed Lazowska, David Molnar, Fabian Monrose, Lincoln Ritter, Avi Rubin, Jason Schultz, Adam Stubblefield, Dan Wallach, and David Wetherall. S. Agarwal was funded by NSF EIA-0321235, University of Washington Animation Research Labs, Washington Research Foundation, Adobe and Microsoft. T. Kohno thanks Cisco Systems Inc. for a gift supporting his research on information leakage and the interactions between compression and encryption.

## References

- [1] Apple ‘It’s Showtime!’ event – live coverage. <http://www.macworld.com/news/2006/09/12/showtime/index.php>.
- [2] D. Balfanz, G. Durfee, R. Grinter, D. Smetters, and P. Stewart. Network-in-a-Box: How to set up a secure wireless network in under a minute. In *13th USENIX Security Symposium*, 2004.
- [3] M. Bellare, A. Boldyreva, A. Desai, and D. Pointcheval. Key-privacy in public-key encryption. In C. Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 566–582. Springer-Verlag, Dec. 2001.
- [4] M. Bellare, A. Desai, E. Jorjipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 394–403. IEEE Computer Society Press, 1997.
- [5] A. Beresford and F. Stajano. Location privacy in pervasive computing. In *IEEE Pervasive Computing*, January 2003.
- [6] BlueTags to install world’s first Bluetooth tracking system, 2003. <http://www.geekzone.co.nz/content.asp?contentid=1070>.
- [7] Bluetooth Spam in Public Places. <http://it.slashdot.org/it/07/01/28/2114253.shtml>.
- [8] Braces – A Bluetooth Tracking Utility. <http://braces.shmoo.com/>.
- [9] D. Chaum and E. van Heyst. Group signatures. In D. W. Davies, editor, *Advances in Cryptology – EUROCRYPT ’91*, volume 547 of *Lecture Notes in Computer Science*, pages 257–265, 1991.
- [10] J. Collins. Lost and Found in Legoland, RFID Journal, 2004. <http://www.rfidjournal.com/article/articleview/921/1/1/>.
- [11] J. Cornwell, I. Fette, G. Hsieh, M. Prabaker, J. Rao, K. Tang, K. Vanica, L. Bauer, L. Cranor, J. Hong, B. McLaren, M. Reiter, and N. Sadeh. User-controllable security and privacy for pervasive computing. In *8th IEEE Workshop on Mobile Computing Systems and Applications (HotMobile 2007)*, 2007.
- [12] Y. Duan and J. Canny. Protecting user data in ubiquitous computing environments: Towards trustworthy environments. In *Workshop on Privacy Enhancing Technology*, 2004.
- [13] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. *ACM SIGMOD Record*, 23(2):419–429, 1994.
- [14] FT2232C Dual USB UART/FIFO IC. <http://www.ftdichip.com/Products/FT2232C.htm>.
- [15] M. Gruteser and D. Grunwald. A methodological assessment of location privacy risks in wireless hotspot networks. In *First International Conference on Security in Pervasive Computing*, 2003.
- [16] M. Gruteser and D. Grunwald. Enhancing location privacy in wireless LAN through disposable interface identifiers: A quantitative analysis. In *ACM Mobile Networks and Applications (MONET)*, volume 10, pages 315–325, Hingham, MA, USA, 2005. Kluwer Academic Publishers.
- [17] M. Haase and M. Handy. BlueTrack – Imperceptible tracking of bluetooth devices. In *UbiComp Poster Proceedings*, 2004.

- [18] T. Hastie, R. Tibshirani, J. Friedman, et al. *The elements of statistical learning: data mining, inference, and prediction*. Springer, 2001.
- [19] J. I. Hong and J. A. Landay. An architecture for privacy-sensitive ubiquitous computing. In *Second International Conference on Mobile Systems, Applications, and Services (Mobisys 2004)*, 2004.
- [20] J. I. Hong and J. A. Landay. Privacy risk models for designing privacy-sensitive ubiquitous computing systems. In *Designing Interactive Systems (DIS2004)*, 2004.
- [21] Y.-C. Hu and H. J. Wang. A framework for location privacy in wireless networks. In *ACM SIGCOMM Asia Workshop*, 2005.
- [22] G. Iachello, I. Smith, S. Consolvo, M. chen, and G. Abowd. Developing privacy guidelines for social location disclosure applications and services. In *Symposium on Usable Privacy and Security*, 2005.
- [23] iPod Connector Female SMD. [http://www.sparkfun.com/commerce/product\\_info.php?products\\_id=8035](http://www.sparkfun.com/commerce/product_info.php?products_id=8035).
- [24] iPod Linux. [http://ipodlinux.org/Dock\\_Connector](http://ipodlinux.org/Dock_Connector).
- [25] A. Jacobs and G. Abowd. A framework for comparing perspectives on privacy and pervasive technologies. In *IEEE Pervasive Computing Magazine. Volume 2, Number 4, October-December*, 2003.
- [26] M. Jakobsson and S. Wetzel. Security weaknesses in bluetooth. In *2001 Conference on Topics in Cryptography*, 2001.
- [27] A. Juels. RFID security and privacy: A research survey. In *IEEE Journal on Selected Areas in Communications*, 2006.
- [28] A. Juels and S. Weis. Authenticating pervasive devices with human protocols. In *25th Annual International Cryptography Conference*, August 2005.
- [29] A. LaMarca, Y. Chawathe, S. Consolvo, J. Hightower, I. Smith, J. Scott, timothy Sohn, J. Howard, J. Hughes, F. Potter, J. Tabert, P. Powledge, G. Borriello, and B. Schilit. Place lab: Device positioning using radio beacons in the wild. In *Third International Conference on Pervasive Computing*, 2005.
- [30] M. Liberatore and B. N. Levine. Inferring the Source of Encrypted HTTP Connections. In *Proc. ACM conference on Computer and Communications Security (CCS)*, October 2006.
- [31] Loca – About Loca. <http://www.loca-lab.org/>.
- [32] Microsofts Zune Delivers Connected Music and Entertainment Experience. <http://www.microsoft.com/presspass/press/2006/sep06/09-14ZuneUnveiling%PR.msp>.
- [33] D. Molnar and D. Wagner. Privacy and security in library RFID issues, practices, and architectures. In *11th ACM Conference on Computer and Communications Security (CCS 2004)*, 2004.
- [34] G. Myles, A. Friday, and N. Davies. Preserving privacy in environments with location-based applications. In *IEEE Pervasive Computing*, 2003.
- [35] Nike and Apple Launch Nike+iPod Sport Kit (for real). <http://www.engadget.com/2006/07/13/nike-and-apple-launch-nike-ipod-sport-kit-for-real/>.
- [36] Nike + iPod Frequently Asked Questions (Technical). <http://docs.info.apple.com/article.html?artnum=303934>. Last accessed on November 12, 2006.
- [37] E. O'Neill, V. Kostakos, T. Kindberg, A. F. gen. Schieck, A. Penn, D. S. Fraser, and T. Jones. Instrumenting the city: Developing methods for observing and understanding the digital cityscape. In *Ubicomp*, 2006.
- [38] S. Orfanidis. *Introduction to Signal Processing*. Inglewood Cliffs. NJ: Prentice-Hall, 1996.
- [39] M. Pels, J. Barhorst, M. Michels, R. Hobo, and J. Barendse. Tracking people using bluetooth: Implications of enabling bluetooth discoverable mode, 2005. Manuscript.
- [40] Sling Media Announces SlingCatcher. [http://us.slingmedia.com/object/io\\_1168286861787.html](http://us.slingmedia.com/object/io_1168286861787.html).
- [41] Sling Media's Newly-Released SlingBox Uses Microsoft Windows Media and Texas Instruments Digital Media Technology to Deliver On-the-Go Entertainment. [http://us.slingmedia.com/object/io\\_1157566629962.html](http://us.slingmedia.com/object/io_1157566629962.html).
- [42] S. A. Weis, S. E. Sarma, R. L. Rivest, and D. W. Engels. Security and Privacy Aspects of Low-Cost Radio Frequency Identification Systems. In *Security in Pervasive Computing*, 2004.
- [43] WireShark. <http://www.wireshark.org>.
- [44] F.-L. Wong and F. Stajano. Location privacy in bluetooth. In *2nd European Workshop on Security and Privacy in Ad hoc and Sensor Networks*, 2005.
- [45] C. Wright, F. Monrose, and G. Masson. On Inferring Application Protocol Behaviors in Encrypted Network Traffic. In *Journal of Machine Learning Research (JMLR): Special issue on Machine Learning for Computer Security*, 2006.
- [46] C. V. Wright, L. Ballard, F. Monrose, and G. M. Masson. Language identification of encrypted VoIP traffic: Alejandro y Roberto or Alice and Bob? In *16th Usenix Security Symposium*, 2007.

# Web-Based Inference Detection

Jessica Staddon<sup>1</sup>, Philippe Golle<sup>1</sup>,  
Bryce Zimny<sup>2\*</sup>

<sup>1</sup> Palo Alto Research Center  
{staddon,pgolle}@parc.com

<sup>2</sup> University of Waterloo  
bzzimny@student.cs.uwaterloo.ca

## Abstract

Newly published data, when combined with existing public knowledge, allows for complex and sometimes unintended inferences. We propose semi-automated tools for detecting these inferences prior to releasing data. Our tools give data owners a fuller understanding of the implications of releasing data and help them adjust the amount of data they release to avoid unwanted inferences.

Our tools first extract salient keywords from the private data intended for release. Then, they issue search queries for documents that match subsets of these keywords, within a reference corpus (such as the public Web) that encapsulates as much of relevant public knowledge as possible. Finally, our tools parse the documents returned by the search queries for keywords not present in the original private data. These additional keywords allow us to automatically estimate the likelihood of certain inferences. Potentially dangerous inferences are flagged for manual review.

We call this new technology Web-based inference control. The paper reports on two experiments which demonstrate early successes of this technology. The first experiment shows the use of our tools to automatically estimate the risk that an anonymous document allows for re-identification of its author. The second experiment shows the use of our tools to detect the risk that a document is linked to a sensitive topic. These experiments, while simple, capture the full complexity of inference detection and illustrate the power of our approach.

## 1 Introduction

Information has never been easier to find. Search engines allow easy access to the vast amounts of information available on the Web. Online data repositories,

newspapers, public records, personal webpages, blogs, etc., make it easy and convenient to look up facts, keep up with events and catch up with people.

On the flip side, information has never been harder to hide. With the help of a search engine or web information integration tool [45], one can easily infer facts, reconstruct events and piece together identities from fragments of information collected from disparate sources. Protecting information requires hiding not only the information itself, but also the myriad of clues that might indirectly lead to it. Doing so is notoriously difficult, as seemingly innocuous information may give away one's secret.

To illustrate the problem, consider a redacted biography [8] (shown in the left-hand side of figure 6) that was released by the FBI. Prior to publication, the biography was redacted to protect the identity of the person whom it describes. All directly identifying information, such as first and last names, was expunged from the biography. The redacted biography contains only keywords that apply to many individuals, such as “half-brother”, “Saudi”, “magnate” and “Yemen”. None of these keywords is particularly identifying on its own, but in aggregate they allow for near-certain identification of Osama Bin Laden. Indeed, a Google search for the query “Saudi magnate half-brother” returns in the top 10 results, pages that are all related to the Bin Laden family. This inference, as well as potentially many others, should be anticipated and countered in a thorough redaction process.

The need to protect secret information from unwanted inferences extends far beyond the FBI. In addition to intelligence agencies and the military, numerous government agencies, businesses and individuals face the problem of insulating their secrets from the information they disclose publicly. In the litigation industry for example, information protected by client-attorney privilege must be redacted from documents prior to disclosure. In the healthcare industry, it is common practice and mandated by some US state laws, to redact sensitive information

\*This work was done while a coop student at the Palo Alto Research Center.

(such as HIV status, drug or alcohol abuse and mental health conditions) from medical records prior to releasing them. Among individuals, anonymous bloggers are a good example of people who seek to ensure that their posts do not disclose their secret (their identity). This is made challenging by the fact that in some cases very little personal information may suffice to infer the blogger's identity. For example, if the second author of this paper were to reveal his first name (Philippe) and mention the first name of his wife (Sanae), then his last name (or at least, a strong candidate for his last name) can be inferred from the first hit returned by the Google query, "Philippe Sanae wedding".

In all these instances, the problem is not access control, but *inference control*. Assuming the existence of mechanisms to control access to a subset of information, the problem is to determine what information can be released publicly without compromising certain secrets, and what subset of the information cannot be released. What makes this problem difficult is the quantity and complexity of inferences that arise when published data is combined with, and interpreted against, the backdrop of public knowledge and outside data.

This paper breaks new ground in considering the problem of inference detection not in a restricted setting (such as, e.g., database tables), but in all its generality. We propose the first all-purpose approach to detecting unwanted inferences. Our approach is based on the observation that the combination of search engines and the Web, which is so well suited to detect inferences, works equally well defensively as offensively. The Web is an excellent proxy for public knowledge, since it encapsulates a large fraction of that knowledge (though certainly not all). Furthermore, the dynamic nature of the Web reflects the dynamic nature of human knowledge and means that the inferences detected today may be different from those drawn yesterday. The likelihood of certain inferences can thus be estimated automatically, at any point in time, by issuing search queries to the Web. Returning to the example of the biography redacted by the FBI, a simple search query could have flagged the risk of re-identification coming from the keywords "Saudi", "magnate" and "half-brother".

The Web is an ideal resource for identifying inferences because keyword search allows for efficient detection of the information that is associated with an individual. Such associations can be just as important in identifying someone as their personal attributes. As an example, consider the fact that the top 2 hits returned by the Google query, "pop singer vogueing"<sup>1</sup> have nothing to do with the singer Madonna, whereas the top 3 hits returned by the Google query, "gay pop singer vogueing"<sup>2</sup> all pertain to Madonna. The attribute "gay" helps to focus the results *not* because it is an attribute of Madonna

(at least not as it is used in the top 3 hits) but rather it is an attribute associated with a large subset of her fanbase. Similarly, the entire first page of hits returned by the query "naltrexone acamprosate" all pertain to alcoholism, not because they are alcoholism symptoms or in some other way part of the definition of alcoholism, but rather they are associated with alcoholism because they are drugs commonly used in its treatment.

We propose generic tools for detecting unwanted inferences automatically using the Web. These tools first extract salient keywords from the private data intended for release. Then, they issue search queries for documents that match subsets of these keywords, within a reference corpus (such as the public Web) that encapsulates as much of relevant public knowledge as possible. Finally, our tools parse the documents returned by the search queries for keywords not present in the original private data. These additional keywords allow us to automatically estimate the likelihood of certain inferences. Potentially dangerous inferences are flagged for manual review. We call this new technology Web-based inference control.

We demonstrate the success of our inference detection tools with two experiments. The first experiment shows the use of our tools to automatically estimate the risk that an anonymous document allows for re-identification of its author. The second experiment shows the use of our tools to detect the risk that a document is linked to a sensitive topic. These experiments, while simple, capture the full complexity of inference detection and illustrate the power of our approach.<sup>3</sup>

**OVERVIEW.** We discuss related work in section 2. We define our models and tools, as well as our basic algorithm for Web-assisted inference detection in section 3. We list a number of potential applications of Web-assisted inference control in section 4. Section 5 describes two experiments that demonstrate the success of our inference control tools. Section 6 provides an example using Web-based inference detection to improve the redaction process. We conclude in section 7.

## 2 Related Work

Our work can be viewed both as a new technique for inference detection and as a new way of leveraging Web search to understand content. There is substantial existing work in both areas, but ours is the first Web-based approach to inference detection. We discuss the most closely related work in these areas below.

**INFERENCE DETECTION.** Most of the previous work on inference detection has focused on database content (see, for example, [33, 21, 43, 19]). Work in this area takes as input the database schema, the data themselves and,

sometimes, relations amongst the attributes of the database that are meant to model the outside knowledge a human may wield in order to infer sensitive information. To the best of our understanding, no systematic method has been demonstrated for integrating this outside knowledge into an inference detection system. Our work seeks to remedy this by demonstrating the use of the Web for this purpose. When coupled with simple keyword extraction, this general technique allows us to detect inference in a variety of unstructured documents.

A particular type of inference allows the identification of an individual. Sweeney looks for such inferences using the Web in [35] where inferences are enabled by numerical values and other attributes characterizable by regular expressions such as SSNs, account numbers and addresses. Sweeney does not consider inferences based on English language words. We use the indexing power of search engines to detect when words, taken together, are closely associated with an individual.

The closely related problem of author identification has also been extensively studied by the machine learning community (see, for example, [25, 11, 24, 34, 20]). The techniques developed generally rely on a training corpus of documents and use specific attributes like self-citations [20] or writing style [25] to identify authors. Our work can be viewed as exploiting a previously unstudied method of author identification, using information authors reveal about themselves to identify them.

Atallah, et al. [2], describe how natural language processing can potentially be used to sanitize sensitive information when the sanitization rules are already known. Our work is focused on using the Web to identify the sanitization rules.

**WEB-ASSISTED QUERY INTERPRETATION.** There is a large body of work on using the Web to improve query results (see, for example, [16, 32, 10]). One of the fundamental ideas that has come out of this area is to use overlap in query results to establish a connection between distinct queries. In contrast, we analyze the content of the query results in order to detect connections between the query terms and an individual or topic.

**WEB-BASED SOCIAL NETWORK ANALYSIS.** Recently, the Web has been used to detect social networks (e.g., [1, 23]). A key idea in this work is using the Web to look for co-occurrences of names and using this to infer a link in a social network. Our techniques can support this type of analysis, when, for example, names in a network when entered as a Web query, yield a name that is not already in the network. However, our techniques are aimed at a broader goal, that is, understanding *all* inferences that can be drawn from a document.

**WEB-ASSISTED CONTENT ANALYSIS AND ANNOTATION.** There is a large body of work on using the Web

to understand and analyze content. Nakov and Hearst [30] have shown the power of using the Web as training data for natural language analysis. Web-assistance for extracting keywords for the purposes of content indexing and annotation is studied in [12, 37, 26]. This work is focused on automated, Web-based tools for understanding the meaning of the text as written, as opposed to the inferences that can be drawn based on the text. That said, in our work we use very simple content analysis tools, and improvements to our approach could involve more sophisticated content analysis tools including Web-based tools such as those developed in these works.

**WEB-BASED DATA AGGREGATION.** Finally, we note that the commercial world is beginning to offer Web-based data aggregation tools (see, for example [14, 13, 31]) for the purposes of tracking competitor behavior, doing market analysis and intelligence gathering. We are not aware of support for pre-production inference control in these offerings, as is the focus of this paper.

### 3 Model and Generic Algorithm

Let  $\mathcal{C}$  denote a private collection of documents that is being considered for public release, and let  $\mathcal{R}$  denote a collection of reference documents. For example, the collection  $\mathcal{C}$  may consist of the blog entries of a writer, and the collection  $\mathcal{R}$  may consist of all documents publicly available on the Web.

Let  $K(\mathcal{C})$  denote all the knowledge that can be computed from the private collection  $\mathcal{C}$ . The set  $K(\mathcal{C})$  informally represents all the statements and facts that can be logically derived from the information contained in the collection  $\mathcal{C}$ . The set  $K(\mathcal{C})$  could in theory be computed with a complete and sound theorem prover given all the axioms in  $\mathcal{C}$ . In practice, such a computation is impossible and we will instead rely on approximate representations of the set  $K(\mathcal{C})$ . Similarly let  $K(\mathcal{R})$  denote all the knowledge that can be computed from the reference collection  $\mathcal{R}$ .

Informally stated, the problem of inference control comes from the fact that the knowledge that can be extracted from the union of the private and reference collections  $K(\mathcal{C} \cup \mathcal{R})$  is typically greater than the union  $K(\mathcal{C}) \cup K(\mathcal{R})$  of what can be extracted separately from  $\mathcal{C}$  and  $\mathcal{R}$ . The inference control problem is to understand and control the difference:

$$\text{Diff}(\mathcal{C}, \mathcal{R}) = K(\mathcal{C} \cup \mathcal{R}) - (K(\mathcal{C}) \cup K(\mathcal{R})).$$

Returning to the Osama Bin Laden example discussed in the introduction, consider the case where the collection  $\mathcal{C}$  consists of the single declassified FBI document [8], and where  $\mathcal{R}$  consists of all information publicly available on the Web. Let  $S$  denote the statement:

“The declassified FBI document is a biography of Osama Bin Laden”. Since the identity of the person to whom the document pertains has been redacted, it is impossible to learn the statement  $S$  from  $\mathcal{C}$  alone, and so  $S \notin K(\mathcal{C})$ . The statement  $S$  is clearly not in  $K(\mathcal{R})$  either since it is impossible to compute from  $\mathcal{R}$  alone a statement about a document that is in  $\mathcal{C}$  but not in  $\mathcal{R}$ . It follows that  $S$  does not belong to  $K(\mathcal{C}) \cup K(\mathcal{R})$ . But, as shown earlier, the statement  $S$  belongs to  $K(\mathcal{C} \cup \mathcal{R})$ . Indeed, we learn from  $\mathcal{C}$  that the document pertains to an individual characterized by the keywords “Saudi”, “magnate”, “half-brothers”, “Yemen”, etc. We learn from  $\mathcal{R}$  that these keywords are closely associated with “Osama Bin Laden”. If we combine these two sources of information, we learn that the statement  $S$  is true with high probability.

It is critical to understand  $\text{Diff}(\mathcal{C}, \mathcal{R})$  prior to publishing the collection  $\mathcal{C}$  of private documents, to ensure that the publication of  $\mathcal{C}$  does not allow for unwanted inferences. The owner of  $\mathcal{C}$  may choose to withhold from publication parts or all of the documents in the collection based on an assessment of the difference  $\text{Diff}(\mathcal{C}, \mathcal{R})$ . Sometimes, the set of sensitive knowledge  $K^*$  that should not be leaked is explicitly specified. In this case, the inference control problem consists more precisely of ensuring that the intersection  $\text{Diff}(\mathcal{C}, \mathcal{R}) \cap K^*$  is empty.

### 3.1 Basic Approach

In this work, we consider the case in which  $\mathcal{C}$  can be any arbitrary collection of documents. In particular, contrary to prior work on inference control in databases, we do not restrict ourselves to private documents formatted according to a well-defined structure. We assume that the collection  $\mathcal{R}$  of public documents consists of all publicly available documents, and that the public Web serves as a good proxy for this collection. Our generic approach to inference detection is based on the following two steps:

1. UNDERSTANDING THE CONTENT OF THE DOCUMENTS IN THE PRIVATE COLLECTION  $\mathcal{C}$ . We employ automated content analysis in order to efficiently extract keywords that capture the content of the document in the collection  $\mathcal{C}$ . A wide array of NLP tools are possible for this process, ranging from simple text extraction to deep linguistic analysis. For the proof-of-concept demonstrations described in section 5, we employ keyword selection via a “term frequency - inverse document frequency” (TF.IDF) calculation, but we note that a deeper linguistic analysis may produce better results.
2. EFFICIENTLY DETERMINING THE INFERENCES THAT CAN BE DRAWN FROM THE COMBINATION OF  $\mathcal{C}$  AND  $\mathcal{R}$ . We issue search queries for documents that

match subsets of the keywords extracted in step 1, within a reference corpus (such as the public Web) that encapsulates as much of relevant public knowledge as possible. Our tools then parse the documents returned by the search queries for keywords not present in the original private data. These additional keywords allow us to automatically estimate the likelihood of certain inferences. Potentially dangerous inferences are flagged for manual review.

### 3.2 Inference Detection Algorithm

In this section, we give a generic description of our inference detection algorithm. This description emphasizes conceptual understanding. Specific instantiations of the inference detection algorithms, tailored to two particular applications, are given in section 5. These instantiations do not realize the full complexity of this general algorithm partly for efficiency reasons and partly because of the attributes of the application. We start with a description of the inputs, outputs and parameters of our generic algorithm.

INPUTS: A private collection of documents  $\mathcal{C} = \{C_1, \dots, C_n\}$ , a collection of reference documents  $\mathcal{R}$  and a list of sensitive keywords  $K^*$  that represent sensitive knowledge.

OUTPUT: A list  $\mathcal{L}$  of inferences that can be drawn from the union of  $\mathcal{C}$  and  $\mathcal{R}$ . Each inference is of the form:

$$(W_1, \dots, W_k) \Rightarrow K_0^*,$$

where  $W_1, \dots, W_k$  are keywords extracted from documents in  $\mathcal{C}$ , and  $K_0^* \subseteq K^*$  is a subset of sensitive keywords. The inference  $(W_1, \dots, W_k) \Rightarrow K_0^*$ , indicates that the keywords  $(W_1, \dots, W_k)$ , found in the collection  $\mathcal{C}$ , together with the knowledge present in  $\mathcal{R}$  allow for inference of the sensitive keywords  $K_0^*$ . The algorithm returns an empty list if it fails to detect any sensitive inference.

PARAMETERS: The algorithm is parameterized by a value  $\alpha$  that controls the depth of the NLP analysis of the documents in  $\mathcal{C}$ , by two values  $\beta$  and  $\gamma$  that control the search depth for documents in  $\mathcal{R}$  that are related to  $\mathcal{C}$ , and finally by a value  $\delta$  that controls the depth of the NLP analysis of the documents retrieved by the search algorithm. The values  $\alpha, \beta, \gamma$  and  $\delta$  are all positive integers. They can be tuned to achieve different trade-offs between the running time of the algorithm and the completeness and quality of inference detection.

UNDERSTANDING THE DOCUMENTS IN  $\mathcal{C}$ . Our basic algorithm uses TF.IDF (term frequency - inverse document frequency, see [28] and section 5.1) to extract from each document  $C_i$  in the collection  $\mathcal{C}$  the top  $\alpha$  keywords

that are most representative of  $C_i$ . Let  $S_i$  denote the set of the top  $\alpha$  keywords extracted from document  $C_i$ , and let  $S = \cup_{i=1}^n S_i$ .

**INFERENCE DETECTION.** The list  $\mathcal{L}$  of inferences is initially empty. We consider in turn every subset  $\mathcal{S}' \subseteq \mathcal{S}$  of size  $|\mathcal{S}'| \leq \beta$ . For every such subset  $\mathcal{S}' = (W_1, \dots, W_k)$ , with  $k \leq \beta$ , we do the following:

1. We use a search engine to retrieve from the collection  $\mathcal{R}$  of reference documents the top  $\gamma$  documents that contain all the keywords  $W_1, \dots, W_k$ .
2. With TF.IDF, we extract the top  $\delta$  keywords from this collection of  $\gamma$  documents. Note that these keywords are extracted from the aggregate collection of  $\gamma$  documents (as if all these documents were concatenated into a single large document), not from each individual document.
3. Let  $K_0^*$  denote the intersection of the  $\delta$  keywords from step 2 with the set  $K^*$  of sensitive keywords. If  $K_0^*$  is non-empty, we add to  $\mathcal{L}$  the inference  $\mathcal{C}' \Rightarrow K_0^*$ .

The algorithm outputs the list  $\mathcal{L}$  and terminates.

### 3.3 Variants of the Algorithm

The algorithm of section 3.2 can be tailored to a variety of applications. Two such applications are discussed in exhaustive detail in section 5. Here, we discuss briefly other possible variants of the basic algorithm.

**DETECTING ALL INFERENCES.** In some applications, the set of sensitive knowledge  $K^*$  may not be known or may not be specified. Instead, the goal is to identify all possible inferences that arise from knowledge of the collection of documents  $\mathcal{C}$  and the reference collection  $\mathcal{R}$ . A simple variation of the algorithm given in 3.2 handles this case. In step 3 of the inference detection phase, we record all inferences instead of only inferences that involve keywords in  $K^*$ . Note that this is equivalent to assuming that the set  $K^*$  of sensitive knowledge consists of all knowledge. The algorithm may also track the number of occurrences of each inference, so that the list  $\mathcal{L}$  can be sorted from most to least frequent inference.

**ALTERNATIVE REPRESENTATION OF SENSITIVE KNOWLEDGE.** The algorithm of section 3.2 assumes that the sensitive knowledge  $K^*$  is given as a set of keywords. Other representations of sensitive knowledge are possible. In some applications for example, sensitive knowledge may consist of a topic (e.g. alcoholism, or sexually transmitted diseases) instead of a list of keywords. To handle this case, we need a pre-processing step which converts a sensitive topic into a list of

sensitive keywords. One way of doing so is to issue a search query for documents in the reference collection  $\mathcal{R}$  that contain the sensitive topic, then use TF.IDF to extract from these documents an expanded set of sensitive keywords.

## 4 Example Applications

This section describes a wide array of potential applications for Web-based inference detection. All these applications are based on the fundamental algorithm of section 3. The first two applications are the subjects of the experiments described in detail in section 5. Experimenting with other applications will be the subject of future work.

**REDACTION OF MEDICAL RECORDS.** Medical records are often released to third parties such as insurance companies, research institutions or legal counsel in the case of malpractice lawsuits. State and federal legislation mandates the redaction of sensitive information from medical records prior to release. For example, all references to drugs and alcohol, mental health and HIV status must typically be redacted. This redaction task is far more complex than it may initially appear. Extensive and up-to-date knowledge of diseases and drugs is required to detect all clues and combinations of clues that may allow for inference of sensitive information. Since this medical information is readily available on public websites, the process of redacting sensitive information from medical records can be partially automated with Web-based inference control. Section 5.3 reports on our experiments with Web-based inference detection for medical redaction.

**PRESERVING INDIVIDUAL ANONYMITY.** Intelligence and other governmental agencies are often forced by law (such as the Freedom of Information Act) to release publicly documents that pertain to a particular individual or group of individuals. To protect the privacy of those concerned, the documents must be released in a form that does not allow for unique identification. This problem is notoriously difficult, because seemingly innocuous information may allow for unique identification, as illustrated by the poorly redacted Osama Bin Laden biography [8] discussed in the introduction. Web-based inference control is perfectly suited to the detection of indirect inferences based on publicly available data. Our tools can be used to determine how much information can be released about a person, entity or event while preserving  $k$ -anonymity, i.e. ensuring that it remains hidden in a group of like-entities of size at least  $k$ , and cannot be identified any more precisely within the group. Section 5.2 reports on our experiments with Web-based inference detection for preserving individual anonymity.

**FORMULATION OF REDACTION RULES.** Our Web-based inference detection tools can also be used to pre-compute a set of redaction rules that is later applied to a collection of private documents. For a large collection of private documents, pre-computing redaction rules may be more efficient than using Web-based inference detection to analyze each and every document. In 1995 for example, executive order 12958 mandated the declassification of large amounts of government data [9] (hundreds of millions of pages). Sensitive portions of documents were to be redacted prior to declassification. The redaction rules were exceedingly complex and formulating them was reportedly nearly as time-consuming as applying them. Web-based inference detection is an appealing approach to automatically expand a small set of seed redaction rules. For example, assuming that the keyword “missile” is sensitive, web-based inference detection could automatically retrieve other keywords related to missiles (e.g. “guidance system”, “ballistics”, “solid fuel”) and add them to the redaction rule.

**PUBLIC IMAGE CONTROL.** This application considers the problem of verifying that a document conforms to the intentions of its author, and does not accidentally reveal private information or information that could easily be misinterpreted or understood in the wrong context. This application, unlike others, does not assume that the set of unwanted inferences is known or explicitly defined. Instead, the goal of this application is to design a broad, general-purpose tool that helps contextualize information and may draw an author’s attention to a broad array of potentially unwanted inferences. For example, Web-based inference detection could alert the author of a blog to the fact that a particular posting contains a combination of keywords that will make the blog appear prominently in the results of some search query. This problem is related to other approaches to public image management, such as [13, 31]. Few technical details have been published about these other approaches, but they do not appear focused on inference detection and control.

**LEAK DETECTION.** This application helps a data owner avoid accidental releases of information that was not previously public. In this application of Web-based inference control, the set of sensitive knowledge  $K^*$  consists of all information that was not previously public. In other words, the release of private data should not add anything to public knowledge. This application may have helped prevent, for example, a recent incident in which Google accidentally released confidential financial information in the notes of a PowerPoint presentation distributed to financial analysts [22].

## 5 Experiments

Our experiments focus on exploring the first two privacy monitor applications of section 4: redaction of medical records and preserving individual anonymity. In testing these ideas, we faced two main challenges that constrained our experimental design. First, and most challenging, was designing relevant experiments that we could execute given available data. The second, more pragmatic, challenge was getting the right tools in place and executing the experiments in a time-efficient manner. We describe each of these challenges, and our approach to meeting them, in more detail below.

### 5.1 Experimental Design Challenges and Tools

Ideally, our idea of Web-based inference detection would be tested on authentic documents for which privacy is a chief concern. For example, a corpus of medical records being prepared for release in response to a subpoena would be ideal for evaluating the ability of our techniques to identify sensitive topics. However, such a corpus is hard to come by for obvious reasons. Similarly, a collection of anonymous blogs would be ideal for testing the ability of our techniques to identify individuals, but such blogs are hard to locate efficiently. Indeed, the excitement over the recently released AOL search data, as illustrated by the quick appearance of tools for mining the data (see, for example, [44, 4]), demonstrates the widespread difficulty in finding data appropriate for vetting data mining technologies, of which our inference detection technology is an example.<sup>4</sup>

Given the difficulties of finding unequivocally sensitive data on which to test our algorithms, we used instead publicly available information about an individual, which we anonymized by removing the individual’s first and last names. In most cases, the public information about the individual, thus anonymized, appeared to be a decent substitute for text that the individual might have authored on their blog or Web page.

All of our experiments rely on Java code we wrote for extracting text from html, on calculation of an extended form of TF.IDF (see definition below) for identifying keywords in documents and on the Google SOAP search API [18] for making Web queries based on those keywords.

Our code for extracting text from html uses standard techniques for removing html tags. Because our experiments involved repeated extractions from similarly formatted html pages (e.g Wikipedia biographies) it was most expedient to write our own code, customized for those pages, rather than retrofitting existing text extraction code such as is available in [3].

As mentioned above, in order to determine if a word is a keyword we use the well known TF.IDF metric (see, for example, [28]). The TF.IDF “rank” of a word in a document is defined with respect to a corpus,  $C$ . We state the definition next.

**Definition 1** Let  $D$  be a document that contains the word  $W$  and is part of a corpus of documents,  $C$ . The **term frequency** (TF) of  $W$  with respect to  $D$  is the number of times  $W$  occurs in  $D$ . The **document frequency** (DF) of  $W$  with respect to the corpus,  $C$ , is the total number of documents in  $C$  that contain the keyword  $W$ . The TF.IDF value associated with  $W$  is the ratio:  $TF/DF$ .

Our code implements a variant of TF.IDF in which we first use the British National Corpus (BNC) [27] to stem lexical tokens (e.g. the tokens “accuse”, “accused”, “accuses” and “accusing” would all be mapped to the stem “accuse”). We then use the BNC again to associate with each token the DF of the corresponding stem (i.e. “accuse” in the earlier example).

As with text extraction from html, there are open source (and commercial) offerings for calculating TF.IDF based on a reference corpus. We did not, however, have a reference corpus on which to base our calculations, and thus opted to write our own code to compute TF.IDF based on the DF values reported in the BNC (which is an excellent model for the English language as a whole, and thus presumably also for text found on the Web).

Our final challenge was experimental run-time. Although we did not invest time optimizing our text extraction code for speed it nevertheless proved remarkably efficient in comparison with the time needed to execute Google queries and download Web pages. In addition, Google states that they place a constraint of 1,000 queries per day for each registered developer on the Google SOAP Search API service [18]. This constraint required us to amass enough Google registrations in order to ensure our experiments could run uninterrupted; in our case, given the varying running times of our experiments, 17 registrations proved enough. The delay caused by query execution and Web page download caused us to modify our algorithms to do a less thorough search for inferences than we had originally intended. These modifications almost certainly cause our algorithms to generate an incomplete set of inferences. However, it is also important to note that despite our efforts, our results contain some links that should have been discarded because they either don’t represent new information (e.g. scrapes of the site from which we extracted keywords) or don’t connect the keywords in our query to the sensitive words in a meaningful way (e.g. an online dictionary covering a broad swath of the English language). Hence, it is possible to improve upon our results by changing the param-

eters of our basic experiments to either do more filtering of the query results or analyze more of the query results and require a majority contain the sensitive word(s).

We describe each experiment in detail below.

## 5.2 Web-based De-anonymization

As discussed in section 4 one of our goals is to demonstrate how keyword extraction can be used to warn the end-user of impending identification. Our inference detection technology accomplishes this by constantly amassing keywords from online content proposed for posting by the user (e.g. blog entries) and issuing Web queries based on those keywords. The user is alerted when the hits returned by those queries return their name, and thus is warned about the risk of posting the content.

We simulated this setting with Wikipedia biographies standing in for user-authored content. We removed the biography subject’s name from the biography and viewed the personal content in the biography as being a condensed version of the information an individual might reveal over many posts to their blog, for example. From these “anonymized” biographies we extracted keywords. Subsets of keywords formed queries to Google. A portion of the returned hits were then searched for the biography subject’s name and a flag was raised when a hit that was not a Wikipedia page contained a mention of the biography’s subject. For efficiency reasons, we limited the portion and number of Web pages that were examined. In more detail, our experiment consists of the following steps:

*Input:* a Wikipedia biography,  $B$ :

1. Extract the subject,  $N$ , of the biography,  $B$ , and parse  $N$  into a first name,  $N_1$ , optional middle name or middle initial,  $N'_1$ , and a last name,  $N_2$  (where  $N_j$  is empty if a name in that position is not given in the biography).<sup>5</sup>
2. Extract the top 20 keywords from the Wikipedia biography,  $B$ , forming the set,  $S_B$ , through the following steps:
  - (a) Extract the text from the html.
  - (b) Calculate the enhanced TF.IDF ranking of each word in the extracted text (section 5.1). If present, remove  $N_1$ ,  $N'_1$  and  $N_2$  from this list, and select the top 20 words from the remaining text as the ordered set,  $S_B$ .
3. For  $x = 20, 19, \dots, 1$ , issue a Google query on the top  $x$  keywords in  $S_B$ . Denote this query by  $Q_x$ . For example, if  $W_1, W_2, W_3$  are the top 3 keywords, the Google query  $Q_3$  is:  $W_1 W_2 W_3$ , with no additional punctuation. Let  $\mathcal{H}_x$  be the set of hits

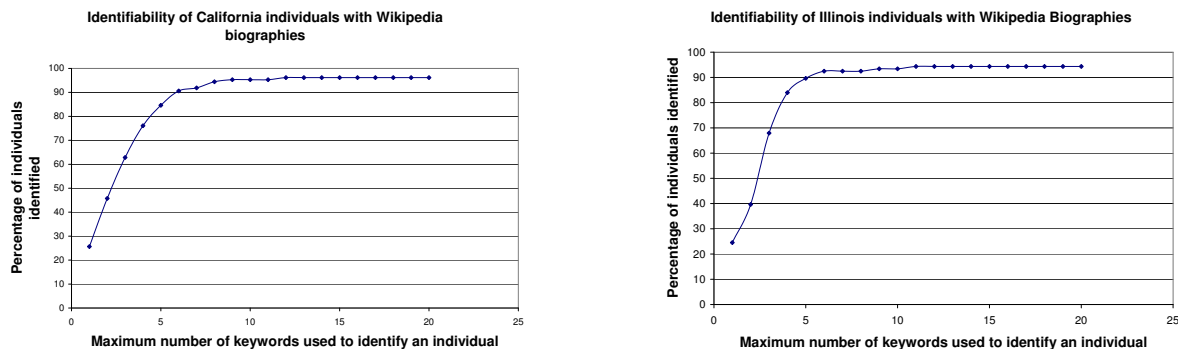


Figure 1: Using 20 keywords per person, extracted from each resident’s Wikipedia biography, the percentage of individuals who were identifiable based on  $x$  keywords or less for  $x = 1, \dots, 20$ . The graph on the left shows results for the 234 biographies of California residents in Wikipedia and the graph on the right shows the results for the 106 biographies of Illinois residents in Wikipedia.

returned by issuing query  $Q_x$  to Google with the restrictions that the hits consist solely of html or text<sup>6</sup> and that no hits from the en.wikipedia.org Web site be returned.

4. Let  $H_{x,1}, H_{x,2}, H_{x,3} \in \mathcal{H}_x$  be the first, second and third hits (respectively) resulting from query  $Q_x$ .<sup>7</sup> For  $x = 20, 19, \dots, 1$ , determine if  $H_{x,1}, H_{x,2}$  and  $H_{x,3}$  contain references to subject,  $N$ , by searching for contiguous occurrences of  $N_1, N'_1$  and  $N_2$  (meaning, no words appear in between the words in a name) within the first 5000 lines of html in each of  $H_{x,1}, H_{x,2}$  and  $H_{x,3}$ . Record any such occurrences.

*Output:*  $S_B$ , each query  $Q_x$  that contains  $N_1, N'_1$  and  $N_2$  contiguously in at least one of the three examined hits, and the url of the particular hit(s).

We ran this test on the 234 biographies of California residents, and the 106 biographies of Illinois residents contained in Wikipedia. The results for both states are shown in Figure 1 and are very similar. In each case, 10 or fewer keywords (extracted from the Wikipedia biography) suffice to identify almost all the individuals. Note that statistics in Figure 1 are based solely on the output of the code, with no human review.

We also include example results (keywords, url, biography subject) in Figure 2. These results illustrate that the associations a person has may be as useful for identifying them as their personal attributes. To highlight one example from the figure, 50% of the first page of hits returned from the Google query “nfl nicole goldman francisco pro” are about O. J. Simpson (including the top 3 hits), but there is no reference to O. J. Simpson in any of

the first page of hits returned by the query “nfl francisco pro”. Hence, the association of O. J. Simpson with his wife (Nicole) and his wife’s boyfriend (Goldman) is very useful to identifying him in the pool of professional football players who once were members of the San Francisco 49ers.

**PERFORMANCE.** In our initial studies, there was wide variation, from a few minutes to over an hour, in the total time it took to process a single biography,  $B$ , depending on the length of the Web pages returned and the number of hits. Hence, in order to efficiently process a sufficiently large number of biographies we restricted the code to only examining the first 5000 lines of html in the returned hits from a given query, and to only search the first 3 hits returned from any given query. With these restrictions, each biography took around 20 minutes to process, with some variation due to differences in biography length. In total, our California experiments took around 78 hours and our Illinois experiments took about 35 hours. Our experimental code does not keep track of the number of queries issued per registration and doing so may yield better performance because switching between registrations occurred only upon receiving a Google SOAP error and so caused some delay.

Our code was not optimized for performance and improvements are certainly possible. In particular, our main slow down came from the text extraction step. One improvement would be to cache Web sites to avoid repeat extractions.

Keywords	URL of Top Hit	Name of Person
campaigned soviets	<a href="http://www.utexas.edu/features/archive/2004/election_policy.html">http://www.utexas.edu/features/archive/2004/election_policy.html</a>	Ronald Reagan
defense contra reagan	<a href="http://www.pbs.org/wgbh/amex/reagan/peopleevents/pande08.html">http://www.pbs.org/wgbh/amex/reagan/peopleevents/pande08.html</a>	Caspar Weinberger
reagan attorney edit pornography	<a href="http://www.sourcewatch.org/index.php?title=Edwin_Meese_III">http://www.sourcewatch.org/index.php?title=Edwin_Meese_III</a>	Edwin Meese
nfl nicole goldman francisco pro	<a href="http://www.brainyhistory.com/years/1997.html">http://www.brainyhistory.com/years/1997.html</a>	O. J. Simpson
kung fu actors	<a href="http://www.amazon.com/Kung-Fu-Complete-Second-Season/dp/B0006BAWYM">http://www.amazon.com/Kung-Fu-Complete-Second-Season/dp/B0006BAWYM</a>	David Carradine
medals medal raid honor aviation	<a href="http://www.voicenet.com/~lpadilla/pearl.html">http://www.voicenet.com/~lpadilla/pearl.html</a>	Jimmy Doolittle
fables chicago indiana	<a href="http://www.indianahistory.org/pop_hist/people/ade.html">http://www.indianahistory.org/pop_hist/people/ade.html</a>	George Ade
wisconsin illinois chicago architect designed	<a href="http://www.greatbuildings.com/architects/Frank_Lloyd_Wright.html">http://www.greatbuildings.com/architects/Frank_Lloyd_Wright.html</a>	Frank Lloyd Wright

Figure 2: Excerpts from our de-anonymization experiments. Each row lists keywords extracted from the Wikipedia biography of an individual (categorized under “California” or “Illinois”), a hit returned by a Google query on those keywords that is one of the top three hits returned and contains the individual’s name, and the name of the individual.

### 5.3 Web-based Sensitive Topic Detection

Another application of Web-based inference detection is the redaction of medical records. As discussed earlier, it is common practice to redact all information about diseases such as HIV/Aids, mental illness, and drug and alcohol abuse, prior to releasing medical records to a third party (such as, e.g., a judge in malpractice litigation). Implementing such protections today relies on the thoroughness of the redaction practitioner to keep abreast of all the medications, physician names, diagnoses and symptoms that might be associated with such conditions and practices. Web-based inference detection can be used to improve the thoroughness of this task by automating the process of identifying the keywords allowing such conditions to be inferred.

To demonstrate how our algorithm can be used in this application, our experiments take as input a page that is viewed as authoritative about a certain disease. In our experiments, we used Wikipedia to supply pages for alcoholism and sexually transmitted diseases (STDs). The text is then extracted from the html, and keywords are identified. To identify keywords that might allow the associated disease to be inferred we then issued Google queries on subsets of keywords and examined the top hit for references to the associated disease. In general, we counted as a reference any mention of the associated disease. The one exception to this rule is that we filtered out some medical term sites since such sites list unrelated medical terms together (for indexing purposes) and we didn’t want such lists to trigger inference results.

In the event that such a reference was found we recorded those keywords as being potentially inference-enabling. In practice, a redaction practitioner might then

use this output to decide what words to redact from medical records before they are released in order to preserve the privacy of the patient.

To gain some confidence in our approach we also used a collection of general medical terms as a “control” and followed the same algorithm. That is, we made Google queries using these medical terms and looked for references to a sensitive disease (STDs and alcoholism) in the returned links. The purpose of this process was to see if the results would differ from those obtained with keywords from the Wikipedia pages about STDs and alcoholism. We expected a distinct difference because the Wikipedia pages should yield keywords more relevant to STDs and alcoholism, and indeed the results indicate that is the case.

The following describes our experiment in more detail.

1. *Input:* An ordered set of sensitive words,  $K^* = \{v_1, \dots, v_b\}$ , for some positive integer  $b$ , and a page,  $B$ .  $B$  is either the Wikipedia page for alcoholism [40], the Wikipedia page for sexually transmitted diseases (STDs) [41] or a “control” page of general medical terms.
  - (a) If  $B$  is a Wikipedia page, extract the top 30 keywords from  $B$ , forming the set  $S_B$ , through the following steps:
    - i. Extract text from html.
    - ii. Calculate the enhanced TF.IDF ranking of each word in the extracted text (section 3). Select the top 30 words as the ordered set,  $S_B = \{W_1, W_2, \dots, W_{30}\}$ .
  - (b) If  $B$  is a medical terms page, extract the terms using code customized for that Web site and

let  $W_B = \{W_1, W_2, \dots, W_{30}\}$  be a subset of 30 terms from that list, where the selection method varies with each run of the experiment (see the results discussion below for the specifics).

- (c) For each pair of words  $\{W_i, W_j\} \in S_B$ , let  $Q_{i,j}$  be the query consisting of just those two words with no additional punctuation and the restriction that no pages from the domain of source page  $B$  be returned, and that all returned pages be text or html (to avoid parsing difficulties). Let  $H_{i,j}$  denote the first hit returned after issuing query  $Q_{i,j}$  to Google, after known medical terms Web sites were removed from the Google results<sup>8</sup>.
- (d) For all  $i, j \in \{1, \dots, 30\}$ ,  $i \neq j$ , and for  $\ell \neq \{1, \dots, b\}$ , search for the string  $v_\ell \in K^*$  in the first 5000 lines of  $H_{i,j}$ . If  $v_\ell$  is found, record  $v_\ell$ ,  $w_i$ ,  $w_j$  and  $H_{i,j}$  and discontinue the search.

2. *Output:* All triples  $(v_\ell, Q_{i,j}, H_{i,j})$  found in step 1, where  $v_\ell$  is in the first 5000 lines of  $H_{i,j}$ .

**RESULTS FOR STD EXPERIMENTS.** We ran the above test on the Wikipedia page about STDs [41],  $B$ , and a selected set,  $B'$ , of 30 keywords from the medical term index [29]. The set  $B'$  was selected by starting at the 49<sup>th</sup> entry in the medical term index and selecting every 400<sup>th</sup> word in order to approximate a random selection of medical terms. As expected, keyword pairs from input  $B$  generated far more hits for STDs ( $306/435 > 70\%$ ) than keyword pairs from  $B'$  ( $108/435 < 25\%$ ). The results are summarized in figure 3.

**RESULTS FOR ALCOHOLISM EXPERIMENTS.** We ran the above test on the Wikipedia page about alcoholism [40],  $B$ , and a selected set,  $B'$ , of 30 keywords from the medical term index [29]. For the run analyzed in Figure 4, the set  $B'$  was selected by starting at the 52<sup>nd</sup> entry in the medical term index and selecting every 100<sup>th</sup> word until 30 were accumulated in order to approximate a random selection of medical terms. As expected, keyword pairs from input  $B$  generated far more hits for alcoholism (47.82%) than  $B'$  (9.43%). In addition, we manually reviewed the URLs that yielded a hit in  $v \in K_{Alc}^*$  for a seemingly innocuous pair of keywords. These results are summarized in figure 4.

**APPLYING THE RESULTS.** When redacting medical records, a redaction practitioner might use the results in figures 3 and 4 to choose content to redact. For example, figure 4 indicates the medications naltrexone and acamprosate should be removed due to their popularity as alcoholism treatments. The words identified as STD-inference enabling are far more ambiguous (e.g.

“transmit”, “infected”). However, for some individuals the very fact that even general terms are frequently associated with sensitive diseases may be enough to justify redaction (e.g. a politician may desire the removal of any “red flag” words). In general though, we think a redaction practitioner could defensibly *not* make it a practice to redact such general terms given their association with other, less sensitive, diseases. This emphasizes that our techniques support semi-automation, but not full automation, of the redaction process.

**PERFORMANCE.** Amortizing the cost of text extraction from the Wikipedia source page over all the queries, determining if each keyword pair yielded a top hit containing a sensitive word took approximately 150 seconds. Hence, each of the experiments in figures 3 and 4 took around 6 hours, since 435 pairs from the Wikipedia page were tested along with 435 pairs from the “control” set of keywords.

As in the de-anonymization experiments, our main time cost was due to the process of text extraction from html. For these experiments caching is likely to significantly improve performance as many of the medical resource sites were visited multiple times.

## 6 Use Scenario: Iterative Redaction

As mentioned in sections 1 and 4, the process of sanitizing documents by removing obviously identifying information like names and social security numbers can be improved by using Web-based inference detection to identify pieces of seemingly innocuous information that can be used to make sensitive inference. To illustrate this idea, we return to the poorly redacted FBI document in the left-hand side of figure 6. Algorithms like those presented in sections 3.2 and 5 can be used to identify sets of keywords that allow for undesired inferences. Some or all of those keywords can then be redacted to improve the sanitization process.

We emphasize that the strategy for redacting based upon the inferences detected by our algorithms is a research problem that is not addressed by this paper. Indeed many strategies are possible. For example, one might redact the minimum set of words (in which case, the redactor seeks to find a minimum set cover for the collection of sets output by the inference detection algorithm). Alternatively, the redactor might be biased in favor of redacting certain parts of speech (e.g. nouns rather than verbs) to enhance readability of the redacted document.

The type of redaction strategy that is employed may influence the Web-based inference detection algorithm. For example, if the goal is to redact the minimum set of words, then it is necessary to consider all possible sets

Summary of STD Experiments		
<b>Input Web Page, <math>B</math>:</b> Wikipedia STD site [41]		
<b>Extracted Keywords, <math>S_B</math>:</b> transmit, sexually, transmitting, transmitted, infection, std, sti, hepatitis, infected, infections, transmission, stis, herpes, viruses, virus, chlamydia, <sup>9</sup> , stds, sexual, disease, hiv, membrane, genital, intercourse, diseases, pmid, hpv, mucous, viral, 2006		
<b>Input Web Page, <math>B'</math>, (“control” page):</b> Medical Terms Site [29]		
<b>Extracted “Control” Keywords, <math>S'_B</math>:</b> Ablation, Ah-Al, Aneurysm, thoracic, Arteria femoralis, Barosinusitis, Bone mineral density, Cancer, larynx, Chain-termination codon, Cockayne syndrome, Cranial nerve IX, Dengue, Disorder, cephalic, ECT, Errors of metabolism, inborn, Fear of nudity, Fracture, comminuted, Gland, thymus, Hecht-Beals syndrome, Hormone, thyroxine, Immunocompetent, Iris melanoma, Laparoscopic, Lung reduction surgery, Medication, clot-dissolving, Mohs surgery, Nasogastric tube, Normoxia, Osteosarcoma, PCR (polymerase chain reaction), Plan B		
<b>Sensitive Keywords, <math>K_{STD}^*</math>:</b> STD, Chancroid, Chlamydia, Donovanosis, Gonorrhea, Lymphogranuloma venereum, Non-gonococcal urethritis, Syphilis, Cytomegalovirus, Hepatitis B, Herpes, HSV, Human Immunodeficiency Virus, HIV, Human papillomavirus, HPV, genital warts, Molluscum, Severe acute respiratory syndrome, SARS, Pubic lice, Scabies, crabs, Trichomoniasis, yeast infection, bacterial vaginosis, trichomonas, mites, nongonococcal urethritis, NGU, molluscum contagiosm virus, MCV, Herpes Simplex Virus, Acquired immunodeficiency syndrome, aids, pubic lice, HTLV, trichomonas, amebiasis, Bacterial Vaginosis, Campylobacter Fetus, Candidiasis, Condyloma Acuminata, Enteric Infections, Genital Mycoplasmas, Genital Warts, Giardiasis, Granuloma Inguinale, Pediculosis Pubis, Salmonella, Shingellosis, vaginitis		
<b>Percentage of words in <math>S_B</math> yielding a top hit containing word(s) in <math>K_{STD}^*</math>:</b> 33.33%		
<b>Percentage of word pairs in <math>S_B</math> yielding a top hit containing word(s) in <math>K_{STD}^*</math>:</b> 70.34%		
<b>Percentage of “control” words in <math>S'_B</math> yielding a top hit containing word(s) in <math>K_{STD}^*</math>:</b> 3.33%		
<b>Percentage of “control” word pairs in <math>S'_B</math> yielding a top hit containing word(s) in <math>K_{STD}^*</math>:</b> 24.83%		
<b>Example keyword pairs from <math>S_B</math> returning a top hit containing a word in <math>K_{STD}^*</math>:</b> <sup>10</sup>		
Keywords	URL of Top Hit	Sensitive Word in Top Hit
transmit, infected	<a href="http://www.rci.rutgers.edu/insects/aids.htm">http://www.rci.rutgers.edu/insects/aids.htm</a>	HIV
transmit, mucous	<a href="http://research.uiowa.edu/animal/?get=empheal">http://research.uiowa.edu/animal/?get=empheal</a>	Herpes
transmitting, viruses	<a href="http://www.cdc.gov/hiv/resources/factsheets/transmission.htm">http://www.cdc.gov/hiv/resources/factsheets/transmission.htm</a>	Hepatitis B
transmitted, viral	<a href="http://www.eurosurveillance.org/em/v10n02/1002-226.asp">http://www.eurosurveillance.org/em/v10n02/1002-226.asp</a>	Hepatitis B
transmitted, infection	<a href="http://www.plannedparenthood.org/sti/">http://www.plannedparenthood.org/sti/</a>	STD
transmitted, disease	<a href="http://www.epigee.org/guide/stds.html">http://www.epigee.org/guide/stds.html</a>	STD
infection, mucous	<a href="http://www.niaid.nih.gov/factsheets/sinusitis.htm">http://www.niaid.nih.gov/factsheets/sinusitis.htm</a>	HIV
infected, disease	<a href="http://www.ama-assn.org/ama/pub/category/1797.html">http://www.ama-assn.org/ama/pub/category/1797.html</a>	HIV
infected, viral	<a href="http://www.merck.com/mmhe/sec17/ch198/ch198a.html">http://www.merck.com/mmhe/sec17/ch198/ch198a.html</a>	Cytomegalovirus
infections, viral	<a href="http://www.nlm.nih.gov/medlineplus/viralinfections.html">http://www.nlm.nih.gov/medlineplus/viralinfections.html</a>	Cytomegalovirus
virus, disease	<a href="http://www.mic.ki.se/Diseases/C02.html">http://www.mic.ki.se/Diseases/C02.html</a>	Cytomegalovirus

Figure 3: Summary of experiments to identify keywords enabling STD inferences.

Summary of Alcoholism Experiments	
<b>Input Web Page, <math>B</math>:</b> Wikipedia Alcoholism site [40]	
<b>Extracted Keywords, <math>S_B</math>:</b> alcoholism, alcohol, drunk, alcoholic, alcoholics, naltrexone, drink, addiction, dependence, detoxification, diagnosed, screening, drinks, moderation, abstinence, 2006, disorder, drinking, behavior, questionnaire, cage, treatment, citation&#160, acamprosate, because, pharmacological, anonymous, extinction, sobriety, dsm	
<b>Input Web Page, <math>B'</math>, (“control” page):</b> Medical Terms Site [29]	
<b>Extracted “Control” Keywords, <math>S'_B</math>:</b> ABO blood group, Alarm clock headache, Ankle-foot orthosis, Ascending aorta, Benign lymphoreticulosis, Breast bone, Carotid-artery stenosis, Chondromalacia patellae, Congenital, Cystic periventricular leukomalacia, Discharge, DX, Enterococcus, Familial Parkinson disease type 5, Fondation Jean Dausset-CEPH, Giant cell pneumonia, Heart attack, Hormone, parathyroid, Impetigo, Itching, Laughing gas, M. intercellulare, Membranous nephropathy, MRSA, Nerve palsy, laryngeal, Oligodendrocyte, Pap Smear, Phagocytosis, Postoperative, Purpura, Henoch-Schonlein	
<b>Sensitive Keywords, <math>K_{Alc}^*</math>:</b> alcoholism, alcoholic(s), alcohol	
<b>Percentage of words in <math>S_B</math> yielding a top hit containing word(s) in <math>K_{Alc}^*</math>:</b> 23.33%	
<b>Percentage of word pairs in <math>S_B</math> yielding a top hit containing word(s) in <math>K_{Alc}^*</math>:</b> 47.82%	
<b>Percentage of “control” words in <math>S'_B</math> yielding a top hit containing word(s) in <math>K_{Alc}^*</math>:</b> 0.00%	
<b>Percentage of “control” word pairs in <math>S'_B</math> yielding a top hit containing word(s) in <math>K_{Alc}^*</math>:</b> 9.43%	
<b>Example word sets from <math>S_B</math> returning a top hit containing a word in <math>K_{Alc}^*</math>:</b> <sup>11</sup>	
Keywords	URL of Top Hit
naltrexone	<a href="http://www.nlm.nih.gov/medlineplus/druginfo/medmaster/a685041.html">http://www.nlm.nih.gov/medlineplus/druginfo/medmaster/a685041.html</a>
acamprosate	<a href="http://www.nlm.nih.gov/medlineplus/druginfo/medmaster/a604028.html">http://www.nlm.nih.gov/medlineplus/druginfo/medmaster/a604028.html</a>
dsm, detoxification	<a href="http://www.aafp.org/afp/20050201/495.html">http://www.aafp.org/afp/20050201/495.html</a> <sup>12</sup>
dsm, detoxification, dependence	<a href="http://www.aafp.org/afp/20050201/495.html">http://www.aafp.org/afp/20050201/495.html</a>

Figure 4: Summary of experiments to identify keywords enabling alcoholism inferences.

Redacted Word(s)	Example Link	Sensitivity of Word(s)
50, 52, 54	<a href="http://multimedia.belointeractive.com/attack/binladen/1004blfamily.html">http://multimedia.belointeractive.com/attack/binladen/1004blfamily.html</a>	Having 50 or more siblings is very characteristic of Osama Bin Laden.
Boston	<a href="http://www.time.com/time/magazine/article/0,9171,1000943,00.html?promoid=googlep">http://www.time.com/time/magazine/article/0,9171,1000943,00.html?promoid=googlep</a>	Many of Osama’s relatives reside in Boston. <sup>13</sup>
magnate	<a href="http://www.outpostoffreedom.com/bin_ladin.htm">http://www.outpostoffreedom.com/bin_ladin.htm</a>	Osama’s father was a building magnate.
denounced, denunciation	<a href="http://www.cairnet.org/html/911statements.html">http://www.cairnet.org/html/911statements.html</a>	A number of groups (including Bin Laden’s family) have denounced his actions.
condemnation	<a href="http://www.usnews.com/usnews/politics/whispers/archive/september2001.htm">http://www.usnews.com/usnews/politics/whispers/archive/september2001.htm</a>	A number of groups (including Bin Laden’s family) have condemned his actions.

Figure 5: Words redacted as a result of Web-based inference detection. Column 1 is the word or words, column 2 is a link using those words output by the algorithm, and column 3 explains why the word(s) are sensitive.

of keywords when looking for inferences. In contrast, if readability is an important concern, then the considered sets might be those favoring certain word types.

What we discuss here is one example of using Web-based inference detection to improve the redaction process. The approach we take is influenced by readability and performance (i.e. speed of the redaction process) but is by no means an optimal approach with respect to either concern. We began by applying some simple redaction rules to the document [8]. Specifically, we removed all location references since our example in section 1 indicated those were important to identifying the biography subject, any dates near September 11, 2001, which is clearly a memorable date, and finally, all citation titles since when paired with the associated publication, these enable the citation articles to be easily retrieved. The resulting redacted document is depicted in figure 6, where grey rectangles indicate the redaction resulting from the rules just described.

Our subsequent redaction proceeded iteratively. At each stage, we extracted the text from the current document, calculated the keywords ordered by the TF.IDF metric and searched for inferences drawn from subsets of a specified number of the top keywords. We then evaluated the output of the algorithm by checking to ensure the produced links did indeed reflect identifying inferences. If a link did not use all the queried keywords in a discussion about Osama Bin Laden then it was deemed invalid. A common source of invalid links were news article titles printed in the side-bar of the link that did not make use of the keywords found in the main body. For example, the query “condone citing prestigious”, yields the top hit [6] (a humor site) because a sidebar links to an article with “Osama” in the title, however, none of the keywords are used in the description of that article.<sup>14</sup>

We incorporated manual review of the links because the current form of our algorithms involves too little content analysis to provide confidence that a returned link reflects a strong connection between the associated keywords and Osama Bin Laden. In addition, given the high security nature of most redaction settings it is unlikely that a purely automated process will ever be accepted.

For those inferences that were found valid, we made redactions to prevent such inferences and repeated this process for the newly redacted document. The following makes the steps we followed precise.

1. Dates near September 11, 2001, titles of all citations and location names were removed from the biography [8].
2. For  $i = 2, \dots, 5$ :
  - (a) We executed Google queries for each  $i$ -tuple in the top  $n_i$  keywords in the biography. The  $n_i$  values were chosen based on performance

constraints as described in section 5.<sup>15</sup> The  $(i, n_i)$  values were:  $(2, 50)$ ,  $(3, 20)$ ,  $(4, 15)$  and  $(5, 13)$ . We concluded with 5-tuples because no valid inferences were found for that run of the algorithm, and only 7% of the links returned by the algorithm run for  $(i, n_i) = (4, 15)$  were valid. For each  $(i, n_i)$  execution of the algorithm we received a list of sets of keywords that were potentially inference-enabling, and the associated top link leading the algorithm to make this conclusion.

- (b) We reviewed the returned links to see if all the corresponding keywords were used in a discussion of Osama Bin Laden. If so, we made a judgement as to which keyword or keywords to remove to remove the inference while preserving readability of the document.
- (c) We incremented  $i$  and returned to step (a) with the current form of the redacted document.

Figure 5 lists the words that were redacted as a result of our Web-based inference detection algorithm. The table also gives an example link output by the algorithm that motivates the redaction and a brief explanation of why the word is sensitive (gained from the manual review of the link(s)). Note that while our algorithm found some document features to be identifying that are unlikely to have been covered by a generic redaction rule (e.g. Osama Bin Laden’s father’s attribute of being a building magnate) it left other, seemingly unusual, attributes (such as Osama Bin Laden potentially being one of 20 children). Since the Web is at best a *proxy* for human knowledge, and our algorithm used the Web in a limited way (i.e. our analysis was limited to a few hits with little NLP use), it seems likely that inferences were missed. Hence, we emphasize that our tool is best used to semi-automate the redaction process.

Finally, we note that the act of redacting information may introduce as well as remove, privacy problems. For example, as noted by Vern Paxson [39], redacting “Boston” without redacting “Globe” may allow the sensitive term “Boston” to be inferred. Our tool suggests “Boston” for redaction, as opposed to “Boston Globe”, because a number of Osama Bin Laden’s relatives reside there, however, acting on this recommendation is problematic precisely because of the difference between the nature of the inference and the document usage of the term. An improved algorithm would understand the use of the term within the document and use this to guide the redaction process.

Our final redacted document is shown in the right hand side of figure 6.

UNCLASSIFIED//FOR OFFICIAL USE ONLY

(U//FOUO) The [redacted] Family b6 -4  
b7C -4

(U//FOUO) [redacted] is a member of a large and wealthy Saudi family. The family patriarch [redacted] came to the kingdom from Hadramout (South Yemen) sometime around 1930.<sup>1</sup>

- In Saudi Arabia, [redacted] father became a construction magnate, completing prestigious projects such as the renovation of the holy mosques in Mecca and Medina. As a result, the [redacted] are a highly respected family both within the Saudi royal household and with the public.

(U//FOUO) There is some confusion as to the total number of [redacted] siblings. b6 -4  
b7C -4

- Some cite that he is the youngest of some 20 sons,<sup>2</sup> while others claim he is the seventh son.<sup>3</sup>
- The total number of his siblings might be 50,<sup>4</sup> 52,<sup>5</sup> or 54.<sup>6</sup> In an interview, [redacted] seemed unsure as well, citing that he had 25 brothers—although he could remember the names of only 20.<sup>7</sup>
- Nearly all of these siblings are half-brothers or half-sisters, as [redacted] father had multiple wives. [redacted] is cited as having only one son.<sup>8</sup>

(U//FOUO) The [redacted] family has denounced [redacted] repeatedly. b6 -2, 4  
b7C -2, 4

- In 1994, the [redacted] family issued a statement expressing its "regret, denunciation and condemnation of all acts that [redacted] may have committed, which we do not condone and we reject."<sup>9</sup> b6 -3
- After the attacks on the US on September 11, 2001, the current head of the family, [redacted] b7C -3

[redacted] b6 -2, 4  
b7C -2, 4

<sup>1</sup> "A Biography of [redacted]" FBI Frontline, 26 September 2001.  
<sup>2</sup> "Child of Privilege Who Champions Holy War," 14 September 2001.  
<sup>3</sup> "Afghanistan: Who He Is and What Makes Him Tick?" Radio Free Europe/Radio Liberty, 14 September 2001.  
<sup>4</sup> "A 'Master Impresario,'" Washington Post, 13 September 2001.  
<sup>5</sup> "Aims to Ride 'Infidels,'" Associated Press, 15 September 2001.  
<sup>6</sup> "Reportedly Says He Has Armed Afghanistan," Boston Globe, 26 September 2001.  
<sup>7</sup> "Denounced by His Family," Independent Television News, 15 September 2001.  
<sup>8</sup> Ibid.

UNCLASSIFIED//FOR OFFICIAL USE ONLY

Saudi Flight-207

Output from using the Web-based inference detection algorithm

UNCLASSIFIED//FOR OFFICIAL USE ONLY

(U//FOUO) The [redacted] Family b6 -4  
b7C -4

(U//FOUO) [redacted] is a member of a large and wealthy [redacted] family. The family patriarch [redacted] came to the kingdom from [redacted] sometime around 1930.<sup>1</sup>

- In [redacted] father became a construction [redacted], completing prestigious projects such as the renovation of the holy mosques in [redacted] and [redacted]. As a result, the [redacted] are a highly respected family both within the [redacted] royal household and with the public.

(U//FOUO) There is some confusion as to the total number of [redacted] siblings. b6 -4  
b7C -4

- Some cite that he is the youngest of some 20 sons,<sup>2</sup> while others claim he is the seventh son.<sup>3</sup>
- The total number of his siblings might be [redacted]. In an interview, [redacted] seemed unsure as well, citing that he had 25 brothers—although he could remember the names of only 20.<sup>7</sup>
- Nearly all of these siblings are half-brothers or half-sisters, as [redacted] father had multiple wives. [redacted] is cited as having only one son.<sup>8</sup>

(U//FOUO) The [redacted] family has [redacted] repeatedly. b6 -2, 4  
b7C -2, 4

- In 1994, the [redacted] family issued a statement expressing its "regret, [redacted] and [redacted] of all acts that [redacted] may have committed, which we do not condone and we reject."<sup>9</sup> b6 -3
- After the attacks on the US on [redacted], the current head of the family, [redacted] b7C -3

[redacted] b6 -2, 4  
b7C -2, 4

<sup>1</sup> [redacted] FBI Frontline, [redacted]  
<sup>2</sup> [redacted] Radio Free Europe/Radio Liberty, [redacted]  
<sup>3</sup> [redacted] Washington Post, [redacted]  
<sup>4</sup> [redacted] Associated Press, [redacted]  
<sup>5</sup> [redacted] Boston Globe, [redacted]  
<sup>6</sup> [redacted] Independent Television News, [redacted]  
<sup>7</sup> Ibid.

UNCLASSIFIED//FOR OFFICIAL USE ONLY

Saudi Flight-207

Figure 6: The left picture shows the original FBI-redacted biography. The right hand side shows the document resulting from using the Web-based inference detection algorithm, where black rectangles represent redactions recommended by the algorithm and grey rectangles are redactions coming from removing dates in 2001, locations and the titles of cited articles (i.e. the grey and black rectangles are redactions made by the authors of this paper).

## 7 Conclusion

We have introduced the notion of using the Web to detect undesired inferences. Our proof-of-concept experiments demonstrate the power of the Web for finding the keywords that are likely to identify a person or topic.

As is to be expected with an initial work, there remains a lot of room for improvement in the algorithms. In particular, to produce an inference detection tool capable of functioning in real-time, as is needed in some applications, improvements already discussed such as Web caching, additional filtering of results to improve precision, and deeper hit analysis to improve recall, are needed. Another avenue for improvement is through deeper content analysis (i.e. beyond keyword extraction). For example, employing a tool capable of deeper semantic analysis such as [15] may allow for both more meaningful extraction of words and phrases for generating queries, and improved analysis of the returned hits for more accurate inference detection. In addition, simple improvements to the content analysis such as better filtering of stop words and html syntax, would create more useful keyword lists.

## Acknowledgement

The authors are very grateful to Richard Chow and Vern Paxson for their help in revising earlier versions of this paper.

## References

- [1] B. Aleman-Meza, M. Nagarajan, C. Ramakrishnan, L. Ding, P. Kolari, A. Sheth, B. Arpinar, A. Joshi and T. Finin. Semantic analytics on social networks: experiences in addressing the problem of conflict of interest detection. *15th International World Wide Web Conference*, 2006.
- [2] M. Atallah, C. McDonough, S. Nirenburg, and V. Raskin. Natural Language Processing for Information Assurance. *Proc. 9th ACM/SIGSAC New Security Paradigms Workshop (NSPW 00)*, pp.51-65, 2000.
- [3] Apache Lucene. <http://lucene.apache.org/java/doc/>
- [4] AOL Keyword Searches. <http://dontdelete.com/default.asp>
- [5] M. Barbaro and T. Zeller. A face is exposed for AOL searcher no. 4417749. *The New York Times*, August 9, 2006.
- [6] <http://www.bongonew.com/layout.php> event
- [7] W. Broad. U. S. Web Archive is Said to Reveal a Nuclear Primer. *The New York Times*, November 3, 2006.
- [8] <http://www.judicialwatch.org/archive/oma.pdf>
- [9] Executive Order 12958, Classified National Security Information. <http://www.dod.mil/eclib/eo.htm>
- [10] B. Davison, D. Deschenes and D. Lewanda. Finding relevant website queries. *Twelfth International World Wide Web Conference*, 2003.
- [11] O. de Vel, A. Anderson, M. Corney and G. Mohay. Mining email content for author identification forensics. *SIGMOD Record*, Vol. 30, No. 4, December 2001.
- [12] Mike Dowman, Valentin Tablan, Hamish Cunningham and Borislav Popov. Web-Assisted Annotation, Semantic Indexing and Search of Television and Radio News. *WWW*, 2005.
- [13] Factiva Insight: Reputation Intelligence. <http://www.factiva.com>
- [14] Fetch Technologies. <http://www.fetch.com>
- [15] GATE: General Architecture for Text Engineering. <http://gate.ac.uk/projects.html>
- [16] N. Glance. Community Search Assistant. *IUI*, 2001.
- [17] P. Golle. Revisiting the Uniqueness of Simple Demographics in the US Population. *Workshop on Privacy in the Electronic Society*, 2006.
- [18] Google SOAP search API. <http://code.google.com/api/soapsearch/>
- [19] J. Hale and S. Sheno. Catalytic inference analysis: detecting inference threats due to knowledge discovery. *IEEE Symposium on Security and Privacy*, 1997.
- [20] S. Hill and F. Provost. The myth of the double-blind review? Author identification using only citations. *SIGKDD Explorations*, 2003.
- [21] T. Hinke. Database inference engine design approach. *Database Security II: Status and Prospects*, 1990.
- [22] D. Jones. Google's PowerPoint blunder was preventable. IR Web Report. <http://www.irwebreport.com/perpective/mar/googleblunder.htm>
- [23] E. Kin, Y. Matsuo, M. Ishizuka. Extracting a social network among entities by web mining. *ISWC '06 Workshop on Web Content Mining with Human Language Technologies*, 2006.
- [24] M. Koppel and J. Schler. Authorship verification as a one-class classification problem. *Proceedings of the 21st International Conference on Machine Learning*, 2004.
- [25] M. Koppel, J. Schler, S. Argamon and E. Messeri. Authorship attribution with thousands of candidate authors. *SIGIR '06*.
- [26] M. Lapata and F. Keller. The Web as a Baseline: Evaluating the Performance of Unsupervised Web-based Models for a Range of NLP Tasks, *HLT-NAACL*, 2004.
- [27] G. Leech, P. Rayson and A. Wilson. *Word frequencies in written and spoken english: based on the British National Corpus*, Longman, London, 2001.
- [28] C. Manning and H. Schutze. Foundations of statistical natural language processing. MIT Press, 1999.
- [29] MedicineNet.com. <http://www.medterm.com/crypt/main/hp.asp>

- [30] P. Nakov and M. Hearst. Using the Web as an Implicit Training Set: Application to Structural Ambiguity Resolution. *HLT-NAACL*, 2005.
- [31] Nstein Technologies. <http://www.nstein.com/pim.asp>
- [32] G. Pant, S. Bradshaw and F. Menczer. Search engine-crawler symbiosis: adapting to community interests. *7th European Conference on Digital Libraries*, 2003.
- [33] X. Qian, M. Stickel, P. Karp, T. Lunt and T. Garvey. Detection and elimination of inference channels in multilevel relational database systems. *IEEE Symposium on Security and Privacy*, 1993.
- [34] M. Steyvers, P. Smyth, M. Rosen-Zvi and T. Griffiths. Probabilistic author-topic models for information discovery. *KDD '04*.
- [35] L. Sweeney. AI Technologies to Defeat Identity Theft Vulnerabilities. *AAAI Spring Symposium on AI Technologies for Homeland Security*, 2005.
- [36] L. Sweeney. Uniqueness of Simple Demographics in the U.S. Population. *LIDAP-WP4*. Carnegie Mellon University, Laboratory for International Data Privacy, Pittsburgh, PA, 2000.
- [37] P. Turney. Coherent Keyphrase Extraction via Web Mining. *IJ-CAI*, 2002.
- [38] Unified Medical Language System. <http://www.umm.edu/glossary/a/index.html>
- [39] Personal communication.
- [40] Wikipedia. Alcoholism. <http://en.wikipedia.org/wiki/Alcoholism>
- [41] Wikipedia. Sexually transmitted disease. [http://en.wikipedia.org/wiki/sexually-transmitted\\_disease](http://en.wikipedia.org/wiki/sexually-transmitted_disease)
- [42] <http://wordweb.info/free/>
- [43] R. Yip and K. Levitt. Data level inference detection in database systems. *IEEE Eleventh Computer Security Foundations Workshop*, 1998.
- [44] D. Zhao and T. Sapp. AOL Search Database. <http://www.aolsearchdatabase.com/>
- [45] <http://www.oominfo.com/>

## Notes

<sup>1</sup><http://www.popandpolitics.com/2005/09/06/and-lite-jazz-singers-shall-lead-the-way/>, [www.popandpolitics.com/2006/10/06/our-paris/](http://www.popandpolitics.com/2006/10/06/our-paris/)

<sup>2</sup>[http://en.wikipedia.org/wiki/Madonna\\_and\\_the\\_gay\\_community](http://en.wikipedia.org/wiki/Madonna_and_the_gay_community), <http://gaybookreviews.info/review/2807/615>, [http://www.youtube.com/results?search\\_type=related&search\\_query=madonna%20oh%20father](http://www.youtube.com/results?search_type=related&search_query=madonna%20oh%20father)

<sup>3</sup>Example results from our experiments appear in section 5. Because of the dynamic nature of the Web, issuing the same queries today may yield somewhat different results.

<sup>4</sup>The AOL data can potentially be used to demonstrate the Web's ability to de-anonymize ([5] may be one such example), which is one of the goals of our algorithms, however because our target application is the protection of English language content, we opted not to vet our algorithms with that data.

<sup>5</sup>The vast majority of the biographies we used identified their subject by both a first and last name with no middle name or initial. Also, name suffixes (e.g. Jr. or annotations made by Wikipedia authors regarding profession), were ignored.

<sup>6</sup>This was done to avoid difficulties parsing non-ascii pages.

<sup>7</sup>These are the first three links that appear on the results page, whether or not one URL is a substring of another.

<sup>8</sup>Here "known site" means any site with "medterm" or "medword" in the URL. As this certainly not sufficient to remove all medical terms sites, we manually reviewed the results before generating the example keyword pairs in Figure 3.

<sup>9</sup>Note this extracted non-word indicates a flaw in our text-from-html extraction algorithm.

<sup>10</sup>In a manual review of the word pairs from  $W'_B$  yielding a top hit containing word(s) in  $K_{STD}^*$ , we did not find any hits using the word pair in a meaningful way in relation to a sensitive word. Rather, the hits generally turned out to be medical term lists.

<sup>11</sup>Since all of our sensitive words pertain to the same topic, alcoholism, we did not record which particular sensitive word was contained in the top hit (if any).

<sup>12</sup>Note this is the 4<sup>th</sup> returned hit, indicating a change in our search strategy would improve recall.

<sup>13</sup>The biography only mentions "Boston" in a citation, so this is a conservative redaction choice.

<sup>14</sup>Alternative metrics for validity are of course possible. For example, a more thorough algorithms might look for shared topic (e.g. the events of September 11, 2001) amongst links, and retain any links pertaining to the most popular topic as valid.

<sup>15</sup>We tended to experience problems communicating with Google when when executing algorithm runs that exceeded 1500 queries, hence we chose values of  $\{n_i\}_i$  that yielded query counts in the range of 1000 – 1500.

# Keep Your Enemies Close: Distance Bounding Against Smartcard Relay Attacks

Saar Drimer and Steven J. Murdoch

*Computer Laboratory, University of Cambridge*

<http://www.cl.cam.ac.uk/users/{sd410, sjm217}>

## Abstract

Modern smartcards, capable of sophisticated cryptography, provide a high assurance of tamper resistance and are thus commonly used in payment applications. Although extracting secrets out of smartcards requires resources beyond the means of many would-be thieves, the manner in which they are used can be exploited for fraud. Cardholders authorize financial transactions by presenting the card and disclosing a PIN to a terminal without any assurance as to the amount being charged or who is to be paid, and have no means of discerning whether the terminal is authentic or not. Even the most advanced smartcards cannot protect customers from being defrauded by the simple relaying of data from one location to another. We describe the development of such an attack, and show results from live experiments on the UK's EMV implementation, *Chip & PIN*. We discuss previously proposed defences, and show that these cannot provide the required security assurances. A new defence based on a distance bounding protocol is described and implemented, which requires only modest alterations to current hardware and software. As far as we are aware, this is the first complete design and implementation of a secure distance bounding protocol. Future smartcard generations could use this design to provide cost-effective resistance to relay attacks, which are a genuine threat to deployed applications. We also discuss the security-economics impact to customers of enhanced authentication mechanisms.

## 1 Introduction

Authentication provides identity assurance for, and of, communicating parties. *Relay*, or *wormhole* attacks allow an adversary to impersonate a participant during an authentication protocol by effectively extending the intended transmission range for which the system was designed. Relay attacks have been described since at least

1976 [13, p75] and are simple to execute as the adversary does not need to know the details of the protocol or break the underlying cryptography. A good example is a relay attack on proximity door-access cards demonstrated by Hancke [16]. To gain access to a locked door, the adversary simply relays the challenges from the door to an authorized card, possibly some distance away, and sends the responses back. The only restriction on the attacker is that the signals arrive at the door and remote card within the allotted time, which Hancke showed to be sufficiently liberal. Another example is wormhole attacks on wireless networks by Hu *et al.* [18]. Despite the existence of such attacks, systems susceptible to them are regularly being deployed. One significant reason is that designers consider relay attacks to be too difficult and costly for attackers to deploy. Section 3 aims to show that relay attacks are indeed practical, using as an example the UK's EMV payment system, *Chip & PIN*. These flaws are demonstrated by an implementation of the relay attack that has been tested on live systems.

Once designers appreciate the risk, the next step in building a secure system is to develop defences. Section 4 describes potential countermeasures to the relay attack and compares their cost and effectiveness. While some, which depend on procedural changes, could be deployed quickly and act as an interim measure, none of the conventional technologies meet our requirements of adequate security at low cost. We thus propose an extension to the smartcard standard, based on a distance bounding protocol, which provides adequate resistance to the relay attack and requires minimal changes to smartcards.

Section 5 describes this countermeasure and its relationship with prior work, describes a circuit design, and evaluates its performance and security properties. We have implemented the protocol on an FPGA and shown it to be an effective defence against very capable adversaries. In addition, the experience of both users and merchants is unchanged, a significant advantage over the other proposals we discuss. The impact of this protocol

on the fraud liability landscape is discussed in Section 6.

Our contributions include the description of the practicalities of relay attacks and our confirmation that deployed systems are vulnerable to them. By designing and testing a prototype system for demonstrating this vulnerability, we show that the attack is feasible and an economically viable threat. Also, we detail the design of a distance bounding protocol for smartcards, discuss implementation issues and present results from both normal operation and under simulated attacks. While papers have previously discussed distance bounding protocols, to the best of our knowledge, this is the first time it has been implemented in practice.

## 2 Background

Contact smartcards, also known as *integrated circuit cards* (ICC), as discussed in this paper, are defined by ISO 7816 [19] (for brevity, our description of the specification will be only to the detail sufficient to illustrate our implementation). The smartcard consists of a sheet of plastic with an integrated circuit, normally a specialized microcontroller, mounted on the reverse of a group of eight contact pads. Current smartcards use only five of these: ground, power, reset, clock are inputs supplied by the card reader, and an additional bi-directional asynchronous serial I/O signal over which the card receives commands and returns its response. Smartcards are designed to operate at clock frequencies between 1 and 5 MHz, with the data rate, unless specified otherwise, of 1/372 of that frequency.

Upon insertion of a smartcard, the terminal first supplies the power and clock followed by de-assertion of reset. The card responds with an *Answer-to-Reset* (ATR), selecting which protocol options it supports, including endianness and polarity, flow control, error correction and data rate. All subsequent communications are initiated by the terminal and consist of a four byte header command with an optional variable-length payload.

### 2.1 Payment environment

There are four parties in the basic payment model: the *cardholder* purchasing the goods or service; the *merchant* supplying the goods or service and who controls the payment terminal; the *issuer bank* is in a contractual relationship with the cardholder and issues their card, and; the *acquirer bank* that is in a contractual relationship with the merchant.

To initiate a transaction, the cardholder presents the merchant with his card and agrees to make the payment in exchange for goods or services. The merchant validates that the card is authentic and that the cardholder is authorized to use it, and sends the transaction details to

the acquirer. The acquirer requests transaction authorization from the issuer over a payment system network (e.g. Mastercard or Visa). If the issuer accepts the transaction, this response is sent back to the merchant via the acquirer and the cardholder is given the good or service. Later, the payment is transferred from the cardholder's account at the issuer to the merchant's account at the acquirer.

In reality, payment systems slightly differ from this simplified description. For this paper's purpose, one notable difference is that the merchant may skip the step of contacting the acquirer to verify the transaction. For smaller retailers, this communication is ordinarily done via a telephone connection, so each authorization request incurs a cost. Thus, for low-risk transactions it may not be necessary to go *online*. Also, if the merchant's terminal cannot make contact with the acquirer, due to the phone line being busy or other technical failure, the merchant may still decide to avoid losing the sale and nevertheless accept the transaction.

### 2.2 Smartcard applications

State-of-the-art smartcards are capable of both symmetric and asymmetric cryptography, have several hundreds of KB of non-volatile tamper-resistant memory, and through secure operating systems may support multiple, mutually un-trusting, applications [3]. Although the potential applications are many, they are most commonly used for authentication of the holder, and more specifically for debit and credit card payment systems, where less sophisticated smartcards are used.

Smartcards have advantages in all three authorization processes discussed above, namely:

**Card authentication:** the card was issued by an acceptable bank, is still valid and the account details have not been modified.

**Cardholder verification:** the customer presenting the card is authorized to use it.

**Transaction authorization:** the customer's account has adequate funds for the transaction.

EMV [15], named after its creators, Europay, Mastercard and Visa, is the primary protocol for debit and credit card payments in Europe, and is known by a variety of different names in the countries where it is deployed (e.g. "Chip & PIN" in the UK). While the following section will introduce the EMV protocol, other payment systems are similar.

In its non-volatile memory, the smartcard may hold account details, cryptographic keys, a *personal identification number* (PIN) and a count of how many consecutive times the PIN has been incorrectly entered.

Cards capable of asymmetric cryptography can cryptographically sign account details under the card's private key to perform card authentication. The merchant's terminal can verify the signature with a public key which is stored on the card along with a certificate signed by the issuer whose key is, in turn, signed by the operator of the payment system network. This method is known as *dynamic data authentication* (DDA) or the variant, *combined data authentication* (CDA).

As the merchants are not trusted with the symmetric keys held by the card, which would enable them to produce forgeries, cards that are only capable of symmetric cryptography cannot be reliably authenticated offline. However, the card can still hold a static signature of account details and corresponding certificate chain. The terminal can authenticate the card by checking this signature, known as *static data authentication* (SDA), but the lack of freshness allows replay attacks to occur.

Cardholder verification is commonly performed by requiring that the cardholder enter their PIN into the merchant's terminal. The PIN is sent to the card which then checks if there have been too many consecutive incorrect guessing attempts; if not, it checks if the PIN was entered correctly. If the terminal or card does not support PIN verification, or the cardholder declines to enter it, the merchant may allow signature verification, or in unattended terminal scenarios, no authentication at all.

The card may hold a history of transactions since it has last communicated with the issuer, and evaluate the risk of authorizing further transactions *offline*; otherwise, the card can request online authorization. In both cases, the card's symmetric keys are used to produce a transaction certificate that is verified by the issuer. Merchants may also force a transaction to be online.

### 2.3 Security goals and threat model

The full threat model of EMV incorporates risk management protocols where the card and terminal negotiate different methods of authenticating cardholders and the conditions for online or offline verification. This decision is reached by considering the transaction value and type (cash-back or goods), the card's record of recent offline transactions and both the card issuer's and merchant's risk perception. This complexity and other features of EMV exist to manage the reality of all parties mistrusting all others (to varying extents). These details are outside the scope of the paper and are further discussed in the EMV specification [15, book 2].

Instead, we assume that the merchant, the banks and customers are honest. We also exclude physical attacks, exploits of software vulnerabilities on both the smartcard and terminal, as well as attacks on the underlying cryptography. Other weaknesses of the EMV system are

known, such as replay attacks on SDA cards as discussed above, and *fallback* attacks which force use of the magnetic stripe, still present on smartcards for backwards compatibility. These weaknesses have been covered elsewhere [1, 4] and are anticipated to be resolved by eventually disabling these legacy features.

In our scenario, the goal of the attacker is to obtain goods or services by charging an unwitting victim who thinks she is paying for something different, at an attacker controlled terminal.

## 3 Relay attacks

Relay attacks were first described by Conway [13, p75], explaining how someone who does not know the rules of chess could beat a Grandmaster. This is possible by challenging two Grandmasters at postal chess and relaying moves between them. While appearing to play a good game, the attacker will either win against one, or draw against both. Desmedt *et al.* [14] showed how such relay attacks could be applied against a challenge-response payment protocol, in the so called "mafia fraud".

We use the mafia-fraud scenario, illustrated in Figure 1, where an unsuspecting restaurant patron, Alice, inserts her smartcard into a terminal in order to pay a \$20 charge, which is presented to her on the display. The terminal looks just like any one of the numerous types of terminals she has used in the past. This particular terminal, however, has had its original circuitry replaced by the waiter, Bob, and instead of being connected to the bank, it is connected to a laptop placed behind the counter. As Alice inserts her card into the counterfeit terminal, Bob sends a message to his accomplice, Carol, who is about to pay \$2000 for a diamond ring at Dave's jewellery shop. Carol inserts a counterfeit card into Dave's terminal, which looks legitimate to Dave, but conceals a wire connected to a laptop in her backpack.

Bob and Carol's laptops are communicating wirelessly using mobile-phones or some other network. The data to and from Dave's terminal is relayed to the restaurant's counterfeit terminal such that the diamond purchasing transaction is placed on Alice's card. The PIN entered by Alice is recorded by the counterfeit terminal and is sent, via a laptop and wireless headset, to Carol who enters it into the genuine terminal when asked. When the transaction is over, the crooks have paid for a diamond ring using Alice's money, who got her meal for free, but will be surprised when her bank statement arrives.

Despite the theoretical risk being documented, EMV is vulnerable to the relay attack, as suggested by Anderson *et al.* [4]. Some believed that engineering difficulties in deployment would make the attack too expensive, or even impossible. The following section will show that

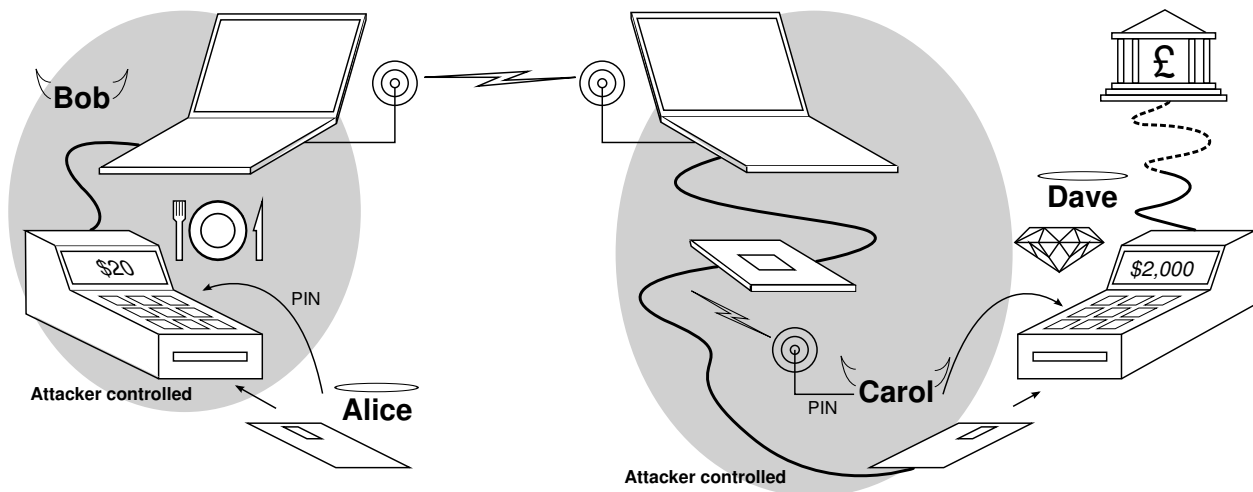


Figure 1: The EMV relay attack. Innocent customer, Alice, pays for lunch by entering her smartcard and PIN into a modified terminal operated by Bob. At approximately the same time, Carol enters her fake card into honest Dave's terminal to purchase a diamond. The transaction from Dave's terminal is relayed wirelessly to Alice's card with the result of Alice unknowingly paying for Carol's diamond.

equipment to implement the attack is readily available, and costs are within the expected returns of fraud.

uine one to the customer even though it lacks the ability to communicate with the bank.

### 3.1 Implementation

This section describes the equipment we used for implementing the relay attack. We chose off-the-shelf components that allowed for fast development rather than miniaturisation or cost-effectiveness. The performance requirements were modest, with the only strict restriction being that our circuit hardware fit within the terminal.

#### 3.1.1 Counterfeit terminal

Chip & PIN terminals are readily available for purchase online and their sale is not restricted. While some are as cheap as \$10, our terminal was obtained for \$50 from eBay and was ideal for our purposes due to its copious internal space. Even if second hand terminals were not so readily available, a plausible counterfeit could be made from scratch as it is only necessary that it appears legitimate to untrained customers.

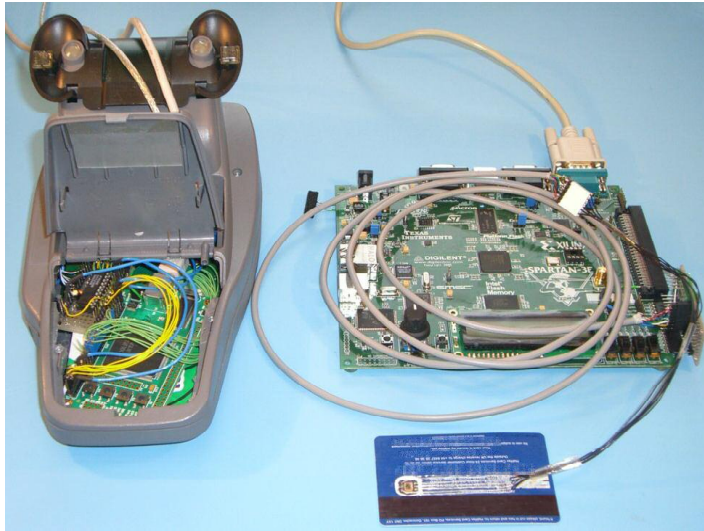
Instead of reverse engineering the existing circuit, we stripped all internal hardware except for the keypad and LCD screen, and replaced it with a \$200 Xilinx Spartan-3 small factor, USB-controlled, development board. We also kept the original smartcard reader slot, but wired its connections to a \$40 USB GemPC Twin reader so we could connect it to the laptop. The result is a terminal with which we can record keypad strokes, display content on the screen and interact with the inserted smart-card. The terminal appears and behaves just like a gen-

#### 3.1.2 Counterfeit card

At the jeweller's, Carol needs to insert a counterfeit card connected to her laptop, into Dave's terminal. We took a genuine Chip & PIN card and ground down the resin-covered wire bonds that connect the chip to the back of the card's pads. With the reverse of the pads exposed, using a soldering iron, we pressed into the plastic thin, flat wires to the edge of the card. This resulted in a card that looked authentic from on the top side, but was actually wired on the back side, as shown in Figure 2. The counterfeit card was then connected through a 1.5 m cable to a \$150 Xilinx Spartan-3E FPGA Starter Kit board to buffer the communications and translate them between the ISO 7816 and RS-232 protocols. Since the FPGA is not 5V tolerant, we use 390  $\Omega$  resistors on the channels that receive data from the card. For the bi-directional I/O channel, we use the Maxim 1740/1 level translator, which costs less than \$2.

#### 3.1.3 Controlling software

The counterfeit terminal and card are controlled by separate laptops via USB and RS-232 interfaces, respectively, using custom software written in Python. The laptops communicate via TCP over 802.11b wireless, although in principle this could be GSM or other wireless protocol. This introduces significant latency, but far less than would be a problem as the timing critical operations



(a) With the exterior intact, the terminal's original internal circuitry was replaced by a small factor FPGA board (left); FPGA based smartcard emulator (right) connected to counterfeit card (front).



(b) Customer's view of terminal. Here, it is playing Tetris, to demonstrate that we have full control of the display and keypad.

Figure 2: Photographs of tampered terminal and counterfeit card.

on the counterfeit card are performed by the FPGA with real-time guarantees.

One complication of selecting an off-the-shelf USB smartcard reader for the counterfeit terminal is that it operates at the *application protocol data unit* (APDU) level and buffers an entire command or response before sending it to the smartcard or the PC. This increases the time between when the genuine terminal sends a command and when the response can be sent; but, as previously mentioned, this is well within tolerances.

This paper only deals with the “T=0” ISO 7816 sub-protocol, as used by all EMV smartcards we have examined. Here, commands are uni-directional, i.e. either the command or response contains a payload but not both. Upon receiving a command code from the genuine terminal, any associated payload will not be sent by the terminal until the card acknowledges the command. The counterfeit card thus cannot tell whether to request a payload (for terminal → card commands) or send the command code to the genuine card immediately (for card → terminal commands).

Were the counterfeit terminal to incorporate a character level card reader, the partial command code could be sent to the genuine card and the result examined to determine the direction, but this is not permissible for APDU level transactions. Hence, the controlling software must be told the direction for each of the 14 command codes. Other than this detail, the relay attack is protocol-agnostic and could be deployed against any ISO 7816 based system.

### 3.2 Procedure and timing

EMV offers a large variety of options, but the generality of the relay attack allows our implementation to account for them all; for simplicity, we will describe the procedure for the common case in the UK. That is, *SDA card authentication* (only the static cryptographic signature of the card details is checked), *online transaction authorization* (the merchant will connect to the issuer to verify that adequate funds are available) and *offline plaintext PIN cardholder verification* (the PIN entered by the cardholder is sent to the card, unencrypted, and the card will check its correctness).

Transaction authorization is accomplished by the card generating an *application cryptogram* (AC), which is authenticated by the card's symmetric key and incorporates transaction details from the terminal, a card transaction counter, and whether the PIN was entered correctly. Thus, the issuing bank can confirm that the genuine card was available and the correct PIN was used. Note that this only requires symmetric cryptography, and so will work even with SDA-only cards, as issued in the UK.

The protocol can be described in six steps:

**Initialization:** The card is powered up and returns the ATR. Then the terminal selects one of the possible payment applications offered by the card.

**Read application data** The terminal requests card details (account number, name, expiration date etc.) and verifies the static signature.

**Cardholder verification:** The cardholder enters their PIN into the merchant's terminal and this is sent to the card for verification. If correct, the card returns a success code, otherwise the cardholder may try again until the maximum number of PIN attempts have been exceeded.

**Generate AC 1:** The terminal requests an *authorization request cryptogram* (ARQC) from the card, which is sent to the issuing bank for verification, which then responds with the *issuer authentication data*.

**External authenticate:** The terminal sends the issuer authentication data to the card.

**Generate AC 2:** The terminal asks the card for a *transaction certificate* (TC) which the card returns to the terminal if, based on the issuer authentication data and other internal state, the transaction is approved. Otherwise, it returns an *application authentication cryptogram* (AAC), signifying the transaction was denied. The TC is recorded by the merchant to demonstrate that it should receive the funds.

This flow imposes some constraints on the relay attack. Firstly, Alice must insert her card before Carol inserts her counterfeit card in order for *initialization* and *read application data* to be performed. Secondly, Alice must enter her PIN before Carol is required to enter it into the genuine terminal. Thirdly, Alice must not remove her card until the *Generate AC 2* stage has occurred. Thus, the two sides of the radio link must be synchronised, but there is significant leeway as Carol can stall until she receives the signal to insert her card.

After that point, the counterfeit card can request extra time from the terminal, before sending the first response, by sending a *null procedure byte* (0x60). The counterfeit terminal can also delay Alice by pretending to dial-up the bank and waiting for authorization until Carol's transaction is complete.

All timing critical sections, such as sending the ATR in response to de-assertion of reset and the encoding/decoding of bytes sent on the I/O, are implemented on the FPGA to ensure a fast enough response. There are wide timing margins between the command and response, so this is managed in software.

### 3.3 Results

We tested our relay setup with a number of different smartcard readers in order to test its robustness. Firstly, we used a VASCO *Chip Authentication Program* (CAP) reader (a similar device, but manufactured by Gemalto, is marketed by the UK bank Barclays as *PINsentry*). This

is a handheld one-time-password generator for use in on-line banking, and implements a subset of the EMV protocol. Specifically, it performs cardholder verification by checking the PIN and requests an application cryptogram, which may be validated online. Our relay setup was able to reliably complete transactions, even when we introduced an extra three seconds of latency between command and response. While the attack we describe in most detail uses the counterfeit card in a retail outlet, a fraudster could equally use a CAP reader to access the victim's online banking. This assumes that the PIN used for CAP is the same as for retail transactions and the criminal knows all other login credentials.

The CAP reader uses a 1 MHz clock to decrease power consumption, but at the cost of slower transactions. We also tested our relay setup with a GemPC Twin reader, which operates at a 4 MHz frequency. The card reader was controlled by our own software, which simulates a Chip & PIN transaction. Here, the relay device also worked without any problems and results were identical to when the card was connected directly to the reader.

Finally, we developed a portable version of the equipment, and took this to a merchant with a live Chip & PIN terminal. With the consent of the merchant and cardholder, we placed a transaction with our counterfeit card in the genuine terminal, and the cardholder's card in the counterfeit terminal. In addition to the commands and responses being relayed, the counterfeit terminal was connected to a laptop which, through voice-synthesis software, read out the PIN to our "Carol". The transaction was completed successfully. One such demonstration of our equipment was shown on the UK consumer rights programme *BBC Watchdog* on 6th February 2007.

### 3.4 Further applications and feasibility

The relay attack is also applicable where "Alice" is not the legitimate card holder, but a thief who has stolen the card and observed the PIN. To frustrate legal investigation and fraud detection measures, criminals commonly use cards in a different country from where they were stolen. Magnetic stripe cards are convenient to use in this way, as the data can be read and sent overseas, to be written on to counterfeit cards. However, chip cards cannot be fully duplicated, so the physical card would need to be mailed, introducing a time window where the cardholder may report the card stolen or lost.

The relay attack can allow fraudsters to avoid this delay by making the card available online using a card reader and a computer connected to the Internet. The fraudster's accomplice in another country could connect to the card remotely and place transactions with a counterfeit one locally. The timing constraints in this scenario are more relaxed as there is no customer expecting

to remove their genuine card. Finally, in certain types of transactions, primarily with unattended terminals, the PIN may not be required, making this attack easier still.

APACS, the UK payment association, say they are unaware of any cases of relay attacks being used against Chip & PIN in the UK [5]. The reason, we believe, is that even though the cost and the technical expertise that are required for implementing the attack are relatively low, there are easier ways to defeat the system. Methods such as card counterfeiting/theft, mail interception, and cardholder impersonation are routinely reported and are more flexible in deployment.

These security holes are gradually being closed, but card fraud remains a lucrative industry – in 2006 £428m ( $\approx$  \$850m) of fraud was suffered by UK banks [6]. Criminals will adapt to the new environment and, to maintain their income, will likely resort to more technically demanding methods, so now is the time to consider how to prevent relay attacks for when that time arrives.

## 4 Defences

The previous section described how feasible it is to deploy relay attacks against Chip & PIN and other smartcard based authorization systems in practice. Thus, system designers must develop mitigation techniques while, for economic consideration, staying within the deployed EMV framework as much as possible.

### 4.1 Non-solutions

In this section we describe a number of solutions that are possible, or have been proposed, against our attack and assess their overall effectiveness.

**Tamper-resistant terminals** A pre-requisite of our relay attack is that Alice will insert her card and enter her PIN into a terminal that relays these details to the remote attacker. The terminal, therefore, must either be tampered with or be completely counterfeit, but still acceptable to cardholders. This implies a potential solution – allow the cardholder to detect malicious terminals so they will refuse to use them. Unfortunately, this cannot be reliably done in practice.

Although terminals do implement internal tamper-responsive measures, when triggered, they only delete keys and other data without leaving visible evidence to the cardholder. Tamper-resistant seals could be inspected by customers, but Johnston *et al.* [21] have shown that many types of seals can be trivially bypassed. It would also be infeasible to give all customers adequate training to detect tampering or counterfeiting of seals. By inducing time-pressure and an awkward physical placement of

the terminal, the attacker can make it extremely difficult for even a diligent customer to check for tampering.

Even if it was possible to produce an effective seal, there are, as of May 2007, 304 VISA approved terminal designs from 88 vendors [24], so cardholders cannot be expected to identify them all. Were there only one terminal design, the use of counterfeit terminals would have to be prevented, which raises the same problems as tamper-resistant seals. Finally, with the large sums of money netted by criminals from card fraud, fabricating plastic parts is well within their budget.

**Imposing additional timing constraints** While relay attacks will induce extra delays between commands being sent by the terminal and responses being received, existing smartcard systems are tolerant to very high latencies. We have successfully tested our relay device after introducing a three second delay into transactions, in addition to the inherent delay of our design. This extra round-trip time could be exploited by an attacker 450 000 km away, assuming that signals propagate at the speed of light. Perhaps, then, attacks could be prevented by requiring that cards reply to commands precisely after a fixed delay. Terminals could then confirm that a card responds to commands promptly and will otherwise reject a transaction.

Other than the *generate AC* command, which includes a terminal nonce, the terminal's behaviour is very predictable. So an attacker could preemptively request these details from the genuine card then send them to the counterfeit card where they are buffered for quick response. Thus, the value of latency as a distance measure can only be exploited at the *generate AC* stages. Furthermore, Clulow *et al.* [12] show how wireless distance bounding protocols, based on channels which were not designed for the purpose, can be circumvented. Their comments apply equally well to wired protocols such as ISO 7816.

To hide the latency introduced by mounting the relay attack, the attacker aims to sample signals early and send signals late, while still maintaining their accuracy. In ISO 7816, cards and terminals are required to sample the signal between the 20% and 80% portion of the bit-time and aim to sample at the 50% point. However, an attacker with sensitive equipment could sample near the beginning, and send their bit late. The attacker then gains 50% of a bit-width in both directions, which at a 5 MHz clock is 37  $\mu$ s, or 11 km.

The attacker could also over-clock the genuine card so the responses are returned more quickly. A DES calculation could take around 100 ms so only a 1% increase would give a 300 km distance advantage. Even if the calculation time was fixed, and only receiving the response from the card could be accelerated, the counterfeit card could preemptively reply with the predictable

11 bytes (2 byte response code, 5 byte *read more* command, 2 byte header and 2 byte counter) each taking 12 bit-widths (start, 8 data bits, stop and 2 bits guard time). At 5 MHz + 1% this gives the attacker 98  $\mu$ s, i.e. 29 km.

One EMV-specific problem is that the contents of the payload in the *generate AC* command are specified by the card in the *card risk management data object list* (CDOL). Although the terminal nonce should be at the end of the message in order to achieve maximum resistance to relay attacks, if the CDOL is not signed, the attacker could substitute the CDOL for one requesting the challenge near the beginning. Upon receiving the challenge from the terminal, the attacker can then send this to the genuine card. Other than the nonce, the rest of the *generate AC* payload is predictable, so the counterfeit terminal can restore the challenge to the correct place, fill in the other fields and send it to the genuine card. Thus, the genuine card will send the correct response, even before the terminal thinks it has finished sending the command. A payload will be roughly 30 bytes, which at 5 MHz gives 27 ms and a 8 035 km distance advantage.

Nevertheless, eliminating needless tolerance to response latency would decrease the options available to the attacker. If it were possible to roll out this modification to terminals as a software upgrade, it might be expedient to plan for this alteration to be quickly deployed in reaction to actual use of the relay attack. While we have described how this countermeasure could be circumvented, attackers who build and test their system with high latency would be forced to re-architect it if the acceptable latency of deployed terminals were decreased without warning.

## 4.2 Procedural improvements

Today, merchants and till operators are accustomed to looking away while customers enter their PIN and seldom handle the card at all, while customers are often recommended not to allow anyone but themselves to handle the card because of card skimming. In the case of relay attacks, this assists the criminal, not the honest customer or merchant. If the merchant examined the card, even superficially, he would detect the relay attack, as we implemented it, by spotting the wires. That said, it is not infeasible that an RFID proximity card could be modified to relay data wirelessly to a local receiver and therefore appear to be a genuine one.

A stronger level of protection can be achieved if, after the transaction is complete, the merchant checks not only that the card presented is legitimate, but also that the embossed card number matches the one on the receipt. In the case of the relay attack, the receipt will show the victim's card number, whereas the counterfeit card will show the original number of the card from before it was

tampered. For these to match, the fraudster must have appropriate blank cards and an embossing machine, in addition to knowing the victim's card number in advance.

Alternatively, a close to real-time attack could still be executed with a portable embossing machine. Existing devices take only a few seconds to print a card and it is feasible that fraudsters can make them portable. The quality of counterfeit cards and embossing need not be high, just sufficient to pass a cursory inspection. More recent smartcards are being issued without embossing, as the carbon-paper payment method is no longer used, making counterfeits even easier to produce. If none of these possibilities are open to the fraudster, repeat customers could be targeted and so creating a wide window of opportunity. In some scenarios, such as unattended Chip & PIN terminals, ATMs, or where the terminal is on the opposite side of a glass barrier, physical card inspections would not be possible; but even where it is, the merchant must be diligent.

Varian [23] argues that if the party who is in the best position to prevent fraud does not have adequate incentives to do so, security suffers. If customers must depend on merchants, who they have no relationship with, for their protection, then there are mismatched incentives. Merchants selling low-marginal-cost products or services (e.g. software or multimedia content), have little desire to carefully check for relay attacks. This is because, in the case of fraud, costs will likely be borne by the customer. Even if the transaction is subsequently reversed when fraud is detected, the merchant has lost only the low marginal cost and the chargeback overhead, but has saved the effort of checking cards.

## 4.3 Hardware alterations

The *electronic attorney* is a trusted device that is brought into the transaction by the customer so that the merchant's terminal does not need to be trusted; this is called the "man-in-the-middle defence", as suggested by Anderson and Bond [2]; trusted devices to protect customers are also discussed by Asokan *et al.* [7]. The device is inserted into the terminal's card slot while the customer inserts their card into the device. The device can display the transaction value as it is parsed from the data sent from the terminal, allowing the customer to verify that she is charged the expected amount. If the customer approves the transaction, she presses a button on the electronic attorney itself, which allows the protocol to proceed. This trusted user interface is necessary, since if a PIN was used as normal, a fraudster could place a legitimate transaction first, which is accepted by the customer, but with knowledge of the PIN a subsequent fraudulent one can be placed. Alternatively, one-time-PINs could be used, but at a cost in usability.

Because the cardholder controls the electronic attorney, and it protects the cardholder's interests, the incentives are properly aligned. Market forces in the business of producing and selling these devices should encourage security improvements. However, this extra device will increase costs, increase complexity and may not be approved of by banking organizations. Additionally, fraudsters may attempt to discourage their use, either explicitly or by arranging the card slot so the use of an electronic attorney is difficult. A variant of the trusted user interface is to integrate a display into the card itself [8].

Another realization of the trusted user interface for payment applications is to integrate the functionality of a smartcard into the customer's mobile phone. This can allow communication with the merchant's terminal using near field communications (NFC) [20]. This approach is already under development and has the advantage of being a customer-controlled device with a large screen and convenient keypad, allowing the merchant's name and transaction value to be shown and once authorized by the user, entry of the PIN. Wireless communications also ease the risk of a malicious merchant arranging the terminal so that the trusted display device is not visible. Although mobile phones are affordable and ubiquitous, they may still not be secure enough for payment applications as they can be, for example, targeted by malware.

## 5 Distance bounding

None of the techniques detailed in Section 4.1 are adequate to completely defeat relay attacks. They are either impractical (tamper-resistant terminals), expensive (adding extra hardware) or circumventable (introducing tighter timing constraints and requiring merchants to check card numbers). Due to the lack of a customer-trusted user interface on the card, there is no way to detect a mismatch between the data displayed on the terminal and the data authorized by the card. However, relay attacks can be foiled if either party can securely establish the position of the card which is authorizing the transaction, relative to the terminal processing it.

Absolute positioning is infeasible due to the cost and form factor requirements of smartcards being incompatible with GPS, and also because the civilian version is not resistant to spoofing [22]. However, it is possible for the terminal to securely establish a maximum distance bound, by measuring the round-trip-time between it and the smartcard; if this time is too long, an alarm would be triggered and the transaction refused. Despite the check being performed at the merchant end, the incentive-compatibility problem is lessened because the distance verification is performed by the terminal and does not depend on the sales assistant being diligent.

The approach of preventing relay attacks by measuring round-trip-time was first proposed by Beth and Desmedt [9] but Brands and Chaum [11] described the first concrete protocol. The cryptographic exchange in our proposal is based on the Hancke-Kuhn protocol [17], because it requires fewer steps, and it is more efficient if there are transmission bit errors compared to Brands-Chaum. However, the Hancke-Kuhn protocol is proposed for ultra-wideband radio (UWB), whereas we require synchronous half-duplex wired transmission.

One characteristic of distance-bounding protocols, unlike most others, is that the physical transmission layer is security-critical and tightly bound to the other layers, so care must be taken when changing the transmission medium. Wired transmission introduces some differences, which must be taken into consideration. Firstly, to avoid circuitry damage or signal corruption, in a wired half duplex transmission, contention (both sides driving the I/O at the same time) must be avoided. Secondly, whereas UWB only permits the transmission of a pulse, wired allows a signal level to be maintained for an extended period of time. Hence, we may skip the initial distance-estimation stage of the Hancke-Kuhn setup and simplify our implementation.

While in this section we will describe our implementation in terms of EMV, implemented to be compatible with ISO 7816, it should be applicable to any wired, half-duplex synchronous serial communication line.

### 5.1 Protocol

In EMV, authentication is only card to terminal so we follow this practise. Following the Hancke-Kuhn terminology, the smartcard is the *prover*,  $P$ , and terminal is the *verifier*,  $V$ . This is also appropriate because the Hancke-Kuhn protocol puts more complexity in the verifier than the prover, and terminals are several orders of magnitude more expensive and capable than the cards. The protocol is described as follows:

Initialization :

$V \rightarrow P : N_V \in \{0, 1\}^a$

$P \rightarrow V : N_P \in \{0, 1\}^a$

$P : (R_i^0 || R_i^1) = H_K(N_V, N_P) \in \{0, 1\}^b$

Rapid bit-exchange :

$V \rightarrow P : C_i \in \{0, 1\}$

$P \rightarrow V : R_i^{C_i} \in \{0, 1\}$

At the start of the *initialization* phase, nonces and parameters are exchanged over a reliable data channel, with timing not being critical.  $N_V$  and  $N_P$  provide freshness to the transaction in order to prevent replay attacks, with the latter preventing a middle-man from running the complete protocol twice between the two phases using the same  $N_V$  and complementary  $C_i$  and thus, obtain

	<i>A</i>	3	8	<i>F</i>	6	<i>D</i>	7	5
$C_i$ :	1010	0011	1000	1111	0110	1101	0111	0101
$R_i^0$ :	$x0x0$	$11xx$	$x011$	$xxxx$	$0xx1$	$xx1x$	$1xxx$	$1x0x$
$R_i^1$ :	$1x0x$	$xx10$	$1xxx$	0001	$x10x$	$01x0$	$x111$	$1x10$
$R_i^{C_i}$ :	1000	1110	1011	0001	0101	0110	1111	1100
	8	<i>E</i>	<i>B</i>	1	5	6	<i>F</i>	<i>C</i>

Table 1: Example of the rapid bit-exchange phase of the distance bounding protocol. For clarity,  $x$  is shown instead of the response bits not sent by the prover. The left most bit is sent first.

both  $R_i^0$  and  $R_i^1$ . The prover produces a MAC under its key,  $K$ , using a keyed pseudo-random function, the result of which is split into two shift registers,  $R_i^0$  and  $R_i^1$ .

In the timing-critical rapid *bit-exchange* phase, the maximum distance between the two participants is determined.  $V$  sends a cryptographically secure pseudorandom single-bit challenge  $C_i$  to  $P$ , which in turn immediately responds with  $R_i^{C_i}$ , the next single-bit response, from the corresponding shift register. A transaction of a 32 bit exchange is shown in Table 1.

If a symmetric key is used, this will require an on-line transaction to verify the result because the terminal does not store  $K$ . If the card has a private/public key pair, a session key can be established and the final challenge-response can also be verified offline. The values  $a$  and  $b$ , the nonce and shift register bit lengths, respectively, are security parameters that are set according to the application and are further discussed in Section 5.5.

This exchange succeeds in measuring distance because it necessitates that a response bit arrive at a certain time after the challenge has been sent. When the protocol execution is complete,  $V$ 's response register,  $R_i^{C_i}$ , is verified by the terminal or bank to determine if the prover is within the allowed distance for the transaction.

## 5.2 Implementation

ISO 7816, our target application, dictates that the smartcard (prover) is a low resource device, and therefore, should have minimal additions in order to keep costs down; this was our prime constraint. The terminal (verifier), on the other hand, is a capable, expensive device that can accommodate moderate changes and additions without adversely affecting its cost. Of course, the scheme must be secure to all attacks devised by a highly capable adversary that can relay signals at the speed of light, is able to ensure perfect signal integrity, and can clock the smartcard at higher frequencies than it was designed for. We assume, however, that this attacker does not have access to the internal operation of the terminal

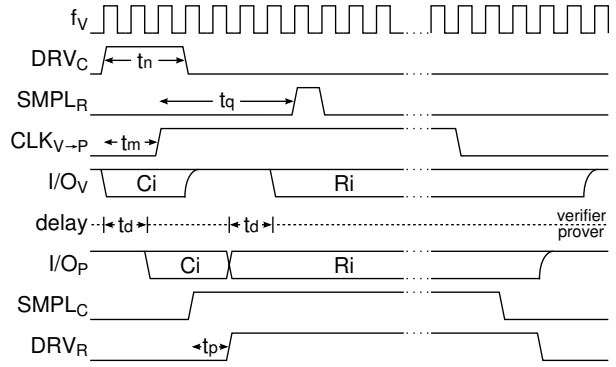


Figure 3: Waveforms of a single bit-exchange of the distance bounding protocol.  $f_V$  is the verifier's clock;  $DRV_C$  drives the challenge on to I/O;  $SMPL_R$  samples the response;  $CLK_{V \rightarrow P}$  is the prover's clock;  $I/O_V$  and  $I/O_P$  are versions of the I/O on each side accounting for the propagation delay  $t_d$ ;  $SMPL_C$  is the received clock that is used to sample the challenge; and  $DRV_R$  drives the response on to the I/O.

and that extracting secret material out of the smartcard, or interfering with its security critical functionality, is not economical considering the returns from the fraud.

## 5.3 Circuit elements and signals

For this section refer to Table 2 for signal names and their function, Figure 4 for the circuit diagram and Figure 3 for the signal waveforms.

**Clocks and frequencies** As opposed to the prover, the verifier may operate at high frequencies. We have implemented the protocol such that one clock cycle of the verifier's operating frequency,  $f_V$ , determines the distance resolution. Since signals cannot travel faster than the speed of light,  $c$ , the upper-bound distance resolution is therefore,  $c/f_V$ . Thus,  $f_V$ , should be chosen to be as high as possible. We selected 200 MHz which allows us a 1.5 m resolution under ideal conditions for the attacker. We have made the prover's operating frequency,  $f_P$ , compatible with any frequencies having a high-time greater than  $t_q + f_V^{-1} + t_d$ , where  $t_q$  defines the time between when the challenge is being driven onto the I/O and when the response is sampled by the verifier;  $t_d$  is the delay between  $V$  and  $P$ . ISO 7816 specifies that the smartcard/prover needs to operate at 1–5 MHz and in order to be compatible, we chose  $f_P = f_V/128 \approx 1.56$  MHz for our implementation.

**Shift registers** The design has four 64 bit shift registers (SR): the verifier's challenge and received response SR's and the prover's two response SR's. The challenge

Signals & timing parameters	Description
$CLK_V, f_V$	Verifier's clock and frequency; determines the distance resolution
$CLK_{V \rightarrow P}, f_P$	Prover's clock and frequency; received from verifier
$DRV_C$	While asserted the challenge is transmitted
$t_n$	Length of time verifier drives the challenge on to the I/O
$SMPL_C$	Prover samples challenge on rising edge
$t_m$	Length of time between assertion of $DRV_C$ to assertion of $CLK_{V \rightarrow P}$
$DRV_R$	Prover transmits response
$t_p$	Amount of delay applied to $SMPL_C$
$SMPL_R$	Verifier samples response on rising edge
$t_q$	Time from assertion of $CLK_{V \rightarrow P}$ to rising edge of $SMPL_R$ ; determines upper bound of prover's distance
$t_d$	Propagation delay through distance $d$

Table 2: Signals and their associated timing parameters.

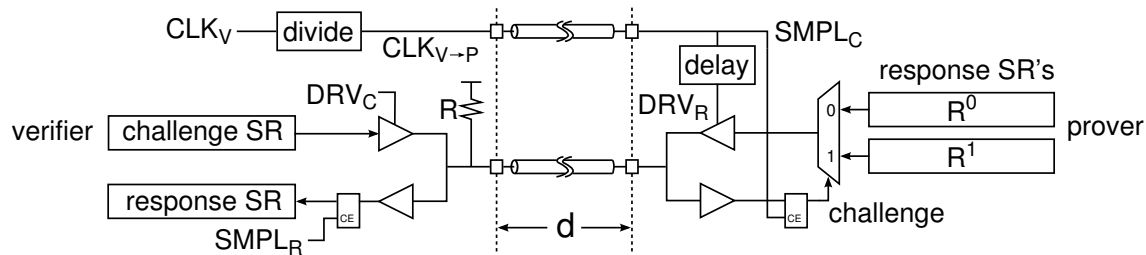


Figure 4: Simplified diagram of the distance bounding circuit.  $DRV_C$  controls when the challenge is put on the I/O line.  $CLK_V$  controls the verifier's circuit; it is divided and is received as  $SMPL_C$  at the prover where it is used to sample the challenge. A delay element produces  $DRV_R$ , which controls when the response is put the I/O, while at the verifier  $SMPL_R$  samples it. The pull-up resistor  $R$  is present to pull the I/O line to a stable state when it is not actively driven by either side.

SR is clocked by  $CLK_V$  and is shifted one clock cycle before it is driven on to the I/O line by  $DRV_C$ . The verifier's response SR is also clocked by  $CLK_V$  and is shifted on the rising edge of  $SMPL_R$ . On the prover side, the SR's are clocked and shifted by  $SMPL_C$ .

**Bi-directional I/O** The verifier and prover communicate using a bi-directional I/O with tri-state buffers at each end. These buffers are controlled by the signals  $DRV_C$  and  $DRV_R$  and are implemented such that only one side drives the I/O line at any given time in order to prevent contention. This is a consequence of adapting the Hancke-Kuhn protocol to a wired medium, and implies that the duration of the challenge must be no longer than necessary, so as to obtain the most accurate distance bound. A pull-up is also present, as with the ISO 7816 specification, to maintain a high state when the line is

not driven by either side. As a side note, if the constraints imposed by ISO 7816 are not to be adhered to, two uni-directional wires for the challenge and response could have been used for easier implementation.

## 5.4 Timing

A timing diagram of a single challenge-response exchange is shown in Figure 3. The circuit shown in Figure 4 was implemented on an FPGA using Verilog (not all peripheral control signals are shown for the sake of clarity). Since we used a single chip, the I/O and clock lines were "looped-back" using various length transmission wires to simulate the distance between the verifier and prover as shown in Figure 5.

The first operation is clocking the challenge shift register (not shown), which is driven on to the I/O line by

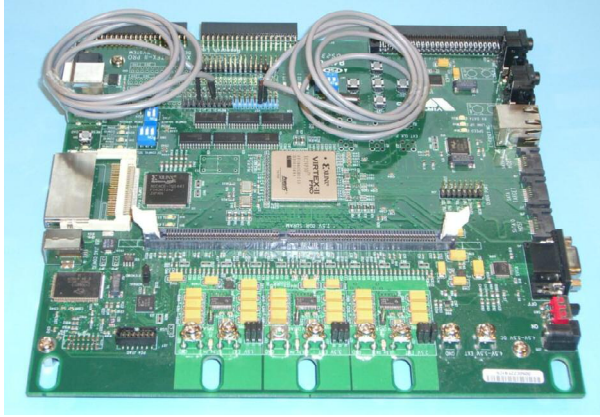


Figure 5: The Xilinx XUP board with a VirtexII-PRO 30 FPGA on which the distance bounding design was implemented. Both verifier and prover reside on the same chip connected only by two same-length transmission lines for I/O and clock (1 m shielded cables are shown).

$DRV_C$  on the following  $f_V$  clock cycle for a  $t_n$  period.  $t_n$  should be made long enough to ensure that the prover can adequately and reliably sample the challenge, and as short as possible to allow the response to be rapidly sent while not causing contention. The clock sent to  $P$ ,  $CLK_{V \rightarrow P}$ , is asserted  $t_m$  after the rising edge of  $DRV_C$ . Both  $CLK_{V \rightarrow P}$  and the I/O line have the same propagation delay,  $t_d$ , and when the clock edge arrives (now called  $SMPL_C$ ), it samples the challenge. The same clock edge also shifts the two response registers, one of which is chosen by a 2:1 multiplexer that is controlled by the sampled challenge.  $DRV_R$  is a delayed replica of  $SMPL_C$ , which is created using a delay element.

The delay,  $t_p$ , allows the response SR signals to shift and propagate through the multiplexer, preventing the intermediate state of the multiplexer from being leaked. Otherwise, the attacker could discover both responses to the previous challenge in the case where  $C_i \neq C_{i-1}$ .  $t_p$  may be very short but should be at least as long as the period from the rising edge of  $SMPL_C$  to when the response emerges from the multiplexer's output; in our implementation, we used deliberately placed routing delays to adjust  $t_p$ , which can be as short as 500 ps. When  $DRV_R$  is asserted, the response is being driven on to the I/O line until the falling edge.

At the verifier, the response is sampled by  $SMPL_R$  after  $t_q$  from the assertion of  $CLK_{V \rightarrow P}$ . The value of  $t_q$  determines the distance measured and should be long enough to account for the propagation delay that the system was designed for (including on-chip and package delays), and short enough to not allow an attacker to be further away than desired, with the minimum value being

$t_p + 2t_d$ . As an improvement,  $t_q$  can be dynamically adjusted between invocations of the protocol allowing the verifier to make decisions based on the measured distance, for example, determine the maximum transaction amount allowed. With a single iteration, the verifier can discover the prover's maximum distance away, but with multiple iterations, the exact distance can be found with a margin of error equal to the signal propagation time during a single clock cycle of the verifier.  $SMPL_R$  may be made to sample on both rising and falling edges of  $f_V$ , effectively doubling the distance resolution without increasing the frequency of operation (other signals may operate this way for tighter timing margins).

If we assume that an attacker can transmit signals at the speed of light and ignore the real-life implications of sending them over long distances, we can determine the theoretical maximum distance between the verifier and prover. A more realistic attacker will need to overcome signal integrity issues that are inherent to any system. We should not, therefore, make it easy for the attacker by designing with liberal timing constraints, and choose the distance between the verifier and prover,  $d$ , to be as short as possible. More importantly, we should carefully design the system to work for that particular distance with very tight margins. For example, the various terminals we have tested were able to transmit/drive a signal through a two meter cable, although the card should at most be a few centimeters away. Weak I/O drivers could be used to degrade the signal when an extension is applied. The value of  $d$  also determines most of the timing parameters of the design, and as we shall see next, the smaller these are, the harder it will be for the attacker to gain an advantage.

## 5.5 Possible attacks on distance bounding

Although, following from our previous assumptions, the attacker cannot get access to any more than half the response bits, there are ways he may extend the distance limit before a terminal will detect the relay attack. This section discusses which options are available, and their effectiveness in evading defences.

**Guessing attack** Following the initialization phase, the attacker can initiate the bit-exchange phase before the genuine terminal has done so. As the attacker does not know the challenge at this stage, he will, on average, guess 50% of the challenge bits correctly and so receive the correct response for those. For the ones where the challenge was guessed incorrectly, the response is effectively random, so there is still a 50% chance that the response will be correct. Therefore the expected success rate of this technique is 75%.

Since our tests show a negligible error rate, the terminal may reject any response with a single bit that is incorrect. In our prototype, where the response registers are 64 bits each, the attacker will succeed with probability  $(\frac{3}{4})^{64} \approx 1$  in  $2^{26}$ . The size of the registers is a security parameter that can be increased according to the application, while the nonces assure that the attacker can only guess once.

**Replay** If the attacker can force the card to perform two protocol runs, with the same nonces used for both, then all bits of the response can be extracted by sending all 1's on the first iteration and all 0's on the second. We resist this attack by selecting the protocol variant mentioned by Hancke and Kuhn [17], where the card adds its own nonce. This is cheap to do within EMV since a transaction counter is already required by the rest of the protocol. If this is not desired then provided the card cannot be clocked at twice its intended frequency, the attacker will not be able to extract all bits in time. This assumes that the time between starting the distance bounding protocol, and the earliest time the high-speed stage can start, is greater than the latter's duration.

**Early bit detection and deferred bit signalling** The card will not sample the terminal's challenge until  $t_{m+d}$  after the challenge is placed on the I/O line. This is to allow an inexpensive card to reliably detect the signal but, as Clulow *et al.* [12] suggest, an attacker who is willing to invest in expensive equipment could, in theory, detect the signal immediately. By manipulating the clock provided to the genuine card, and using high-quality signal drivers, the challenge could be sent to the card with less of a delay.

Similarly, the terminal will wait  $t_q$  between sending the challenge and sampling the response, to allow for the round trip signal propagation time, and wait until the response signal has stabilized. Again, with superior equipment the response could be sent from the card just before the terminal samples. The attacker, however, cannot do so any earlier than  $t_p$  after the card has sampled the challenge, and the response appears on the I/O.

**Delay-line manipulation** The card may include the value of  $t_p$  in its signed data, so the attacker cannot make the terminal believe that the value is larger than the card's specification. However, the attacker might be able to reduce the delay, for example by cooling the card. If it can be reduced to the point that the multiplexer or latch has not settled, then both potential responses may be placed on to the I/O line, violating our assumptions.

However, if the circuit is arranged so that the delay will be reduced only if the reaction of the challenge latch

and multiplexer is improved accordingly, the response will still be sent out prematurely. This gives the attacker extra time, so should be prevented. If temperature compensated delay lines are not economic, then they should be as short as possible to reduce this effect.

In fact,  $t_p$  may be so small, even less than 1 ns, that the terminal could just assume it would be zero. This will mean that the terminal will believe all cards are slightly further away than they really are, but will avoid the value of  $t_p$  having to be included in the signed data.

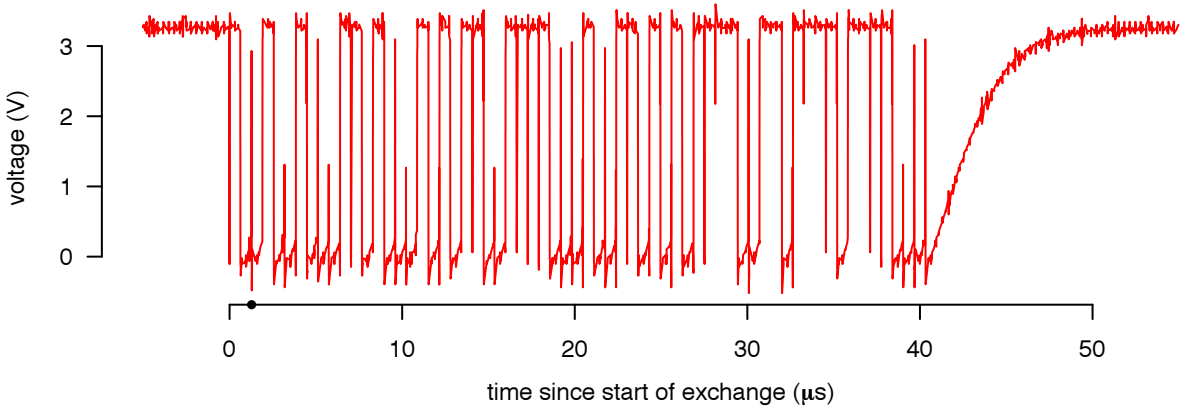
**Combined attacks** For an attacker to gain a better than 1 in  $2^{26}$  probability of succeeding in the challenge response protocol, the relay attack must take less than  $t_{m+q}$  time. In practice, an attacker will not be able to sample or drive the I/O line instantaneously and the radio-link transceiver or long wires will introduce latency, so the attacker would need to be much closer than this limit. A production implementation on an ASIC would be able to give better security guarantees and be designed to tighter specifications than were available on the FPGA for our prototype.

## 5.6 Results

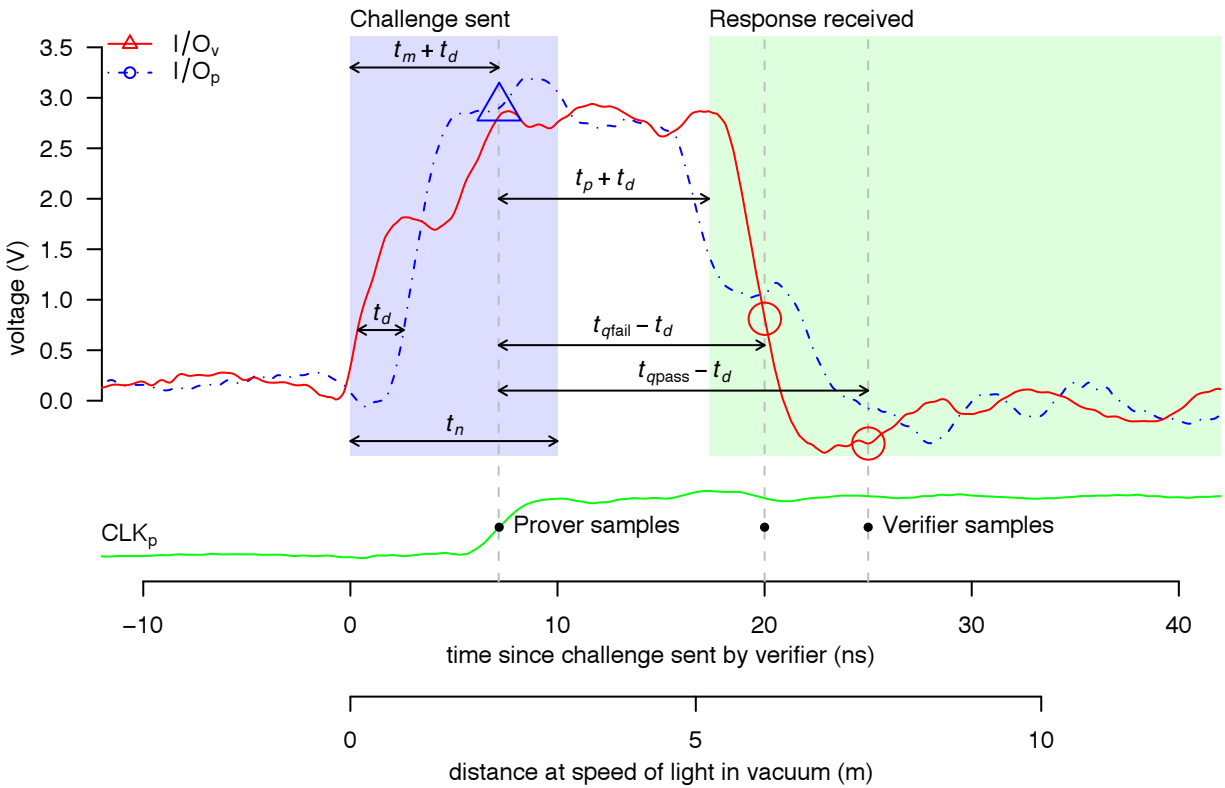
We have developed a versatile implementation that requires only modest modification to currently deployed designs. Our distance bounding scheme was successfully implemented and tested on an FPGA for 2.0, 1.0, and 0.3 meter transmission lengths, although it can be modified to work for any distance and tailored to any end application. Oscilloscope traces of a single bit challenge-response exchange over a 50  $\Omega$ , 30 cm printed circuit board transmission line are shown in Figure 6. In this case, the challenge is 1 and the response is 0 with indicators where SMPL<sub>R</sub> has sampled the response. The first, after  $t_{qfail} = 15$  ns has sampled too early while the second,  $t_{qpass} = 20$  ns, which is a single period of  $f_V$  later, has correctly sampled the 0 as the response. The delay  $t_d = 2.16$  ns, can also be seen and is, of course, due to the length of the transmission line. If the attacker exploited all possible attacks previously discussed and was able to transmit signals at  $c$ , he would need to be within approximately 6 m, although the actual distance would be shorter for a realistic attacker.

## 5.7 Costs

The FPGA design of both the verifier and prover as shown in Figure 5 consumes 37 flip-flops and 93 look-up tables: 64 for logic, 13 route-throughs, and 16 as shift registers (4 cascaded 16-bit LUTs for each), which is extremely compact, and consumes well under 0.5% of the resources available on our FPGA. However, it is



(a)  $I/O_V$  trace of a 64-bit exchange with position of (b) indicated by • on the  $x$  axis.



(b) Single bit exchange, challenge is 1 and response is 0.

Figure 6: Oscilloscope trace from the bit-exchange phase of the distance bounding protocol. Delay is introduced by a 30 cm transmission line between the verifier and prover. Timing parameters are  $t_n = 10$  ns,  $t_m = 5$  ns,  $t_p = 8$  ns. Two values of  $t_q$  are shown, one where the bit was correctly received  $t_{qpass} = 20$  ns and one where it was not,  $t_{qfail} = 15$  ns.  $t_d$  was measured to be 2.16 ns which over a 30 cm wire corresponds to propagation velocity of  $1.4 \times 10^8$  m/s. Note that before the challenge is sent, the trace is slowly rising above ground level; this is the effect of the pull-up resistor as also seen in (a) after the protocol completes. The shown signals were probed at the FPGA I/Os and do not precisely represent when they actually appear inside of it. For example, the FPGA I/O introduces 3–5 ns delay to the signal so in actuality the FPGA will “see” the falling edge shown in (b) slightly after what is represented in the figure. On-chip delay also affects the design and is not shown, but must be accounted for.

difficult to estimate the cost of an ASIC implementation with these figures as there is no reliable conversion technique between FPGA resource utilization and ASIC transistor count, especially since the above numbers are for the core functions, without the supporting circuitry. It is also hard to estimate the cost in currency because that changes rapidly with time, production volume, fabrication process, and many other factors, so we will describe it relative to the resources currently used.

As mentioned, we have made every effort to minimize the circuitry that needs to be added to the smartcard while being more liberal with the terminal, although for both the additions can be considered minor. For the smartcard, new commands for initiating the initialization phase need to be added as well as two shift registers and a state machine for operating the rapid bit-exchange. Considering that smartcards already have a few thousand memory cells, this can be considered a minor addition, especially given that they need to operate at the existing low frequencies of 1–5 MHz. For the initialization phase, existing circuits can be used such as the DES engine for producing the content of the response registers. The card's transaction counter may be used for the nonce,  $N_p$ .

As for the terminals, their internal operating frequency is unknown to us, but it is unlikely that it is high enough to achieve good distance resolution. Therefore, a capable processor and some additional components are required, such as a high quality oscillator. As an alternative to high frequencies, or when designing for very short distances, delay lines could be used instead of operating on clock edges. The distance bounding circuitry would need to be added to the terminal's main processor, which consists of two shift registers and slightly more involved control code than the smartcard's.

We have described the added cost in terms of hardware but the added time per transaction and the need to communicate with the bank, refused transactions due to failure, re-issuing cards, and so on, may amount to substantial costs. Only the banks involved have access to all the necessary information needed to make a reasonable estimate of these overheads.

## 6 Discussion

The distance bounding protocol we have proposed will detect attempted relay attacks but requires the banks to produce cards and terminals that support the protocol. However, the person being defrauded is the cardholder, Alice, who must use the cards and terminals she is given, so has no trusted user interface and no way to protect herself. As mentioned in Section 4.2, this incentive mismatch may be detrimental to the cardholder's security. For instance, until all terminals support the distance bounding protocol, the issuer can select whether to fall-

back to the current protocol that is vulnerable to relay-attacks. Under existing UK practice, the customer is liable for PIN-verified fraudulent transactions [10], so the issuer may elect to accept fallback transactions knowing that the cardholder is carrying the risk.

A further problem of the distance bounding protocol is the lack of non-repudiation: for a third party to verify that a relay attack was not in progress, the merchant's terminal must be trusted to correctly report the round-trip latency. Thus, if a customer claims that a transaction is fraudulent, then even if the distance bounding protocol is recorded to have succeeded, there remains the possibility that the terminal has been tampered with. It falls on the acquirer to mandate tamper-resistant terminals, but although the payment network may require that all members implement appropriate protections, the customer is only represented indirectly by the issuer.

So while inexpensive yet strong technical solutions, such as distance bounding, do exist they must be deployed as part of an appropriate liability framework to fully realize their benefits. The current situation, where customers are liable for fraud, yet powerless to verify whether a terminal is genuine, is clearly unfair. If the power of banking institutions is too great to alter the entrenched notion of customer liability, then measures that put the cardholder in a position of control, such as the *electronic attorney* [2], despite being more expensive, may be the most appropriate solution. However, customers should exercise caution before accepting these options as the second-order effect of the customer being able to detect attacks could be to make them reasonably liable for any fraud which is nevertheless perpetrated.

## 7 Conclusion

This paper described relay attacks and how they can be applied to exploit smartcard-based payment systems. A prototype was built and shown to be successful against the Chip & PIN payment system deployed in the UK. This consisted of creating a fake terminal and custom hardware to allow the relaying of information between the participating parties. We suggested procedural improvements to the acceptance of Chip & PIN transactions, which would provide a short-term defence, but these could be circumvented by a plausible attacker. We then developed the first implementation of a distance bounding defence against these relay attacks and showed it to be the most robust solution. Our implementation was designed to be appealing for adoption in the next generation of smartcards by tailoring the design to the EMV framework.

Future work may include implementing a wireless variant of the protocol, mutual distance bound establishment and customizing the system to other applications.

## Acknowledgements

Saar Drimer is funded by Xilinx, Inc. Steven J. Murdoch is funded by the OpenNet Initiative. Markus Kuhn has provided us with both stimulating discussions and advice, along with hardware. Xilinx donated hardware and development software. We also thank Ross Anderson, Mike Bond, Richard Clayton, Frank Stajano, Robert Watson, Ford-Long Wong, anonymous reviewers for their valuable suggestions, and the merchants who allowed us to test and demonstrate our attack.

## References

- [1] ADIDA, B., BOND, M., CLULOW, J., LIN, A., MURDOCH, S. J., ANDERSON, R. J., AND RIVEST, R. L. Phish and chips (traditional and new recipes for attacking EMV). In *Security Protocols Workshop* (Cambridge, England, March 2006), LNCS, Springer (to appear). <http://www.cl.cam.ac.uk/~rja14/Papers/Phish-and-Chips.pdf>.
- [2] ANDERSON, R., AND BOND, M. The man in the middle defence. In *Security Protocols Workshop* (Cambridge, England, March 2006), Springer (to appear). <http://www.cl.cam.ac.uk/~rja14/Papers/Man-in-the-Middle-Defence.pdf>.
- [3] ANDERSON, R., BOND, M., CLULOW, J., AND SKOROBOGATOV, S. Cryptographic processors—a survey. *Proceedings of the IEEE* 94, 2 (February 2006), 357–369.
- [4] ANDERSON, R., BOND, M., AND MURDOCH, S. J. Chip and spin, March 2005. <http://www.chipandspin.co.uk/spin.pdf>.
- [5] APACS. APACS response to BBC Watchdog and chip and PIN. Press release, February 2007. [http://www.apacs.org.uk/media\\_centre/press/07\\_06\\_02.html](http://www.apacs.org.uk/media_centre/press/07_06_02.html).
- [6] APACS. Card fraud losses continue to fall. Press release, March 2007. [http://www.apacs.org.uk/media\\_centre/press/07\\_14\\_03.html](http://www.apacs.org.uk/media_centre/press/07_14_03.html).
- [7] ASOKAN, N., DEBAR, H., STEINER, M., AND WAIDNER, M. Authenticating public terminals. *Computer Networks* 31, 9 (1999), 861–870.
- [8] AVESO. Display enabled smart cards. <http://www.avesodisplays.com/>.
- [9] BETH, T., AND DESMEDT, Y. Identification tokens – or: Solving the chess grandmaster problem. In *CRYPTO* (1990), vol. 537 of LNCS, Springer, pp. 169–177.
- [10] BOHM, N., BROWN, I., AND GLADMAN, B. Electronic commerce: Who carries the risk of fraud? *The Journal of Information, Law and Technology*, 3 (October 2000). [http://www2.warwick.ac.uk/fac/soc/law/elj/jilt/2000\\_3/bohm/](http://www2.warwick.ac.uk/fac/soc/law/elj/jilt/2000_3/bohm/).
- [11] BRANDS, S., AND CHAUM, D. Distance-bounding protocols. In *EUROCRYPT '93: Workshop on the theory and application of cryptographic techniques on Advances in cryptology* (May 1993), T. Hellese, Ed., vol. 765 of LNCS, Springer, pp. 344–359.
- [12] CLULOW, J., HANCKE, G. P., KUHN, M. G., AND MOORE, T. So near and yet so far: Distance-bounding attacks in wireless networks. In *Security and Privacy in Ad-hoc and Sensor Networks* (Hamburg, Germany, September 2006), L. Buttyan, V. Gligor, and D. Westhoff, Eds., vol. 4357 of LNCS, Springer.
- [13] CONWAY, J. H. *On Numbers and Games*. Academic Press, 1976.
- [14] DESMEDT, Y., GOUTIER, C., AND BENGIO, S. Special uses and abuses of the Fiat-Shamir passport protocol. In *Advances in Cryptology – CRYPTO '87: Proceedings* (1987), vol. 293 of LNCS, Springer, p. 21.
- [15] EMVCO, LLC. *EMV 4.1*, June 2004. <http://www.emvco.com/>.
- [16] HANCKE, G. A practical relay attack on ISO 14443 proximity cards, 2005. <http://www.cl.cam.ac.uk/~gh275/relay.pdf>.
- [17] HANCKE, G. P., AND KUHN, M. G. An RFID distance bounding protocol. In *SECURECOMM '05: Proceedings of the First International Conference on Security and Privacy for Emerging Areas in Communications Networks* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 67–73.
- [18] HU, Y.-C., PERRIG, A., AND JOHNSON, D. Wormhole attacks in wireless networks. *IEEE Journal on Selected Areas in Communications (JSAC)* 24, 2 (February 2006).
- [19] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 7816-3:2006 Identification cards – Integrated circuit cards – Part 3: Cards with contacts – Electrical interface and transmission protocols*, 3 ed., October 2006.
- [20] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 18092:2004 Information technology – Telecommunications and information exchange between systems – Near Field Communication – Interface and Protocol (NFCIP-1)*, 1 ed., January 2007.
- [21] JOHNSTON, R. G., GARCIA, A. R., AND PACHECO, A. N. Efficacy of tamper-indicating devices. *Journal of Homeland Security* (April 2002).
- [22] KUHN, M. G. An asymmetric security mechanism for navigation signals. In *Information Hiding* (Toronto, Canada, May 2004), J. Fridrich, Ed., no. 3200 in LNCS, Springer, pp. 239–252.
- [23] VARIAN, H. R. Managing online security risks. *New York Times*. 1 June, 2000. <http://www.ischool.berkeley.edu/~hal/people/hal/NYTimes/2000-06-01.html>.
- [24] VISA INTERNATIONAL SERVICE ASSOCIATION. Approved PIN entry devices, May 2007. <http://partnernetwork.visa.com/dv/pin/pedapprovallist.jsp>.

# Human-Seeded Attacks and Exploiting Hot-Spots in Graphical Passwords<sup>1</sup>

Julie Thorpe and P.C. van Oorschot  
*School of Computer Science, Carleton University*  
{jthorpe, paulv}@scs.carleton.ca

## Abstract

Although motivated by both usability and security concerns, the existing literature on click-based graphical password schemes using a single background image (e.g., PassPoints) has focused largely on usability. We examine the security of such schemes, including the impact of different background images, and strategies for guessing user passwords. We report on both short- and long-term user studies: one lab-controlled, involving 43 users and 17 diverse images, and the other a field test of 223 user accounts. We provide empirical evidence that popular points (hot-spots) do exist for many images, and explore two different types of attack to exploit this hot-spotting: (1) a “human-seeded” attack based on harvesting click-points from a small set of users, and (2) an entirely automated attack based on image processing techniques. Our most effective attacks are generated by harvesting password data from a small set of users to attack other targets. These attacks can guess 36% of user passwords within  $2^{31}$  guesses (or 12% within  $2^{16}$  guesses) in one instance, and 20% within  $2^{33}$  guesses (or 10% within  $2^{18}$  guesses) in a second instance. We perform an image-processing attack by implementing and adapting a bottom-up model of visual attention, resulting in a purely automated tool that can guess up to 30% of user passwords in  $2^{35}$  guesses for some instances, but under 3% on others. Our results suggest that these graphical password schemes appear to be at least as susceptible to offline attack as the traditional text passwords they were proposed to replace.

## 1 Introduction

The bane of password authentication using text-based passwords is that users choose passwords which are easy to remember, which generally translates into passwords that are easily guessed. Thus even when the size of a password space may be theoretically “large enough”

(in terms of number of possible passwords), the *effective* password space from which many users actually choose passwords is far smaller. Predictable patterns, largely due to usability and memory issues, thus allow successful search by variations of exhaustive guessing attacks. Forcing users to use “random” or other non-meaningful passwords results in usability problems. As an alternative, graphical password schemes require that a user remembers an image (or parts thereof) in place of a word. They have been largely motivated by the well-documented human ability to remember pictures better than words [25], and implied promises that the password spaces of various image-based schemes are not only sufficiently large to resist guessing attacks, but that the effective password spaces are also sufficiently large. The latter, however, is not well established.

Among the graphical password schemes proposed to date, one that has received considerable attention in the research literature is PassPoints [45, 46, 47]. It and other click-based graphical password schemes [18, 4, 31, 37] require a user to log in by clicking a sequence of points on a single background image. Usability studies have been performed to determine the optimal amount of error tolerance [46], login and creation times, error rates, and general perception [45, 47]. An important remaining question for such schemes is: how *secure* are they? This issue remains largely unaddressed, despite speculation that the security of these schemes likely suffers from hot-spots – areas of an image that are more probable than others for users to click. Indeed, the impact of hot-spots has been downplayed (e.g., see [45, Section 7]). In this paper, we focus on a security analysis of an implementation with the same parameters as used in a recent PassPoints publication [47]. A usability analysis of this implementation is presented in a separate paper [6].

We confirm the existence of hot-spots through empirical studies, and show that some images are more susceptible to hot-spotting than others. We also explore the security impact of hot-spots, including a number of strate-

gies for exploiting them under an offline model similar to that used by Ballard et al. [1]. Our work involves two user studies. The first (lab) study used 17 diverse images (four used in previous studies [46], and 13 of our own chosen to represent a range of detail). We collected graphical passwords for 32-40 users per image in a lab setting, and found hot-spots on all images even from this relatively small sample size; some images had significantly more hot-spots than others. In the second (field) study involving 223 user accounts over a minimum of seven weeks, we explore two of these images in greater depth. We analyzed our lab study data using formal measures of security to make an informed decision of which two images to use in the field study. Our goal was to give PassPoints the best chance we could (in terms of anticipated security), by using one highly ranked image, and another mid-ranked image also used in previous PassPoints studies.

We implement and evaluate two types of attack: human-seeded and purely automated. Our human-seeded attack is based on harvesting password data from a small number of users to attack passwords from a larger set of users. We seed various dictionaries with the passwords collected in our lab study, and apply them to guess the passwords from our long-term field study. Our results demonstrate that this style of attack is quite effective against this type of graphical password: it correctly guessed 36% of user passwords within  $2^{31}$  guesses (or 12% within  $2^{16}$  guesses) on one image, and 20% within  $2^{33}$  guesses (or 10% within  $2^{18}$  guesses) on a second image. We implement and adapt a combination of image processing methods in an attempt to predict user choice, and employ them as tools to expedite guessing attacks on the user study passwords. The attack works quite well on some images, cracking up to 30% of passwords, but less than 3% on others within  $2^{35}$  guesses. These results give an early signal that image processing can be a relevant threat, particularly as better methods emerge.

Our contributions include the first in-depth study of hot-spots in click-based (or cued-recall) graphical passwords schemes and their impact on security through two separate user studies: one lab-controlled and the other a field test. We propose the modification and use of image processing methods to expedite guessing attacks, and evaluate our implementation against the images used in our studies. Our implementation is based on Itti et al.'s [17] model of bottom-up visual attention and corner detection, which allowed successful guessing attacks on some images, even with relatively naive dictionary strategies. Our most interesting contribution is applying a human-seeded attack strategy, by harvesting password data in a lab setting from small sets of users, to attack other field study passwords. Our human-seeded attack strategy for cued-recall graphical passwords is sim-

ilar to Davis et al.'s attack [8] against recognition-based graphical passwords; notable differences include a more straightforward dictionary generation method, and that our seed data is from a separate population and (short-term) setting.

The remainder of this paper is organized as follows. Section 2 provides background and terminology. Section 3 presents our lab-controlled user study, and an analysis of observed hot-spots and the distribution of user click-points. Section 4 presents results on the larger (field) user study, and of our password harvesting attacks. Section 5 explores our use of image processing methods to expedite guessing attacks on the 17 images from the first user study and the two from the second user study. Related work is briefly discussed in Section 6. Section 7 provides further discussion and concluding remarks.

## 2 Background and Terminology

Click-based graphical passwords require users to log in by clicking a sequence of points on a single background image. Many variations are possible (see Section 6), depending on what points a user is allowed to select. We study click-based graphical passwords by allowing clicks anywhere on the image (i.e., PassPoints-style). We believe that most findings related to hot-spots in this style will apply to other variations using the same images, as the “interesting” clickable areas are still present.

We use the following terminology. Assume a user chooses a given click-point  $c$  as part of their password. The *tolerable error* or *tolerance*  $t$  is the error allowed for a click-point entered on a subsequent login to be accepted as  $c$ . This defines a *tolerance region* (*T-region*) centered on  $c$ , which for our implementation using  $t = 9$  pixels, is a  $19 \times 19$  pixel square. A *cluster* is a set of one or more click-points that lie within a T-region. The number of click-points belonging to a cluster is its *size*. A *hot-spot* is indicated by a cluster that is large, relative to the number of users in a given sample. To aid visualization and indicate relative sizes for clusters of size at least two, on figures we sometimes represent the underlying cluster by a shaded circle or *halo* with halo diameter proportional to its size. An *alphabet* is a set of distinct T-regions; our implementation, using  $451 \times 331$  pixel images, results in an alphabet of  $m = 414$  T-regions. Using passwords composed of 5-clicks, on an alphabet of size 414 provides the system with only a 43-bit full theoretical password space; we discuss the implications of this in Section 7.

### 3 Lab Study and Clustering Analysis

Here we report on the results of a university-approved 43-user study of click-based graphical passwords in a controlled lab environment. Each user session was conducted individually and lasted about one hour. Participants were all university students who were not studying (or experts in) computer security. Each user was asked to create a click-based graphical password on 17 different images (some of these are reproduced herein; others are available from the authors). Four of the images are from a previous click-based graphical password study by Wiedenbeck et al. [46]; the other 13 were selected to provide a range of values based on two image processing measures that we expected to reflect the amount of detail: the number of segments found from image segmentation [11] and the number of corners found from corner detection [16]. Seven of the 13 images were chosen to be those we “intuitively” believed would encourage fewer hot-spots; this is in addition to the four chosen in earlier research [46] using intuition (no further details were provided on their image selection methodology).

**EXPERIMENTAL DETAILS.** We implemented a web-based experiment. Each user was provided a brief explanation of what click-based graphical passwords are, and given two images to practice creating and confirming such passwords. To keep the parameters as consistent as possible with previous usability experiments of such passwords [47], we used  $d = 5$  click-points for each password, an image size of  $451 \times 331$  pixels, and a  $19 \times 19$  pixel square of error tolerance. Wiedenbeck et al. [47] used a tolerance of  $20 \times 20$ , allowing 10 pixels of tolerated error on one side and 9 on the other. To keep the error tolerance consistent on all sides, we approximate this error tolerance using  $19 \times 19$ . Users were instructed to choose a password by clicking on 5 points, with no two the same. Although the software did not enforce this condition, subsequent analysis showed that the effect on the resulting cluster sizes was negligible for all images except *pcb*; for more details, see caption of Figure 1. We did not assume a specific encoding scheme (e.g., robust discretization [3] or other grid-based methods); the concept of hot-spots and user choice of click-points is general enough to apply across all encoding schemes. To allow for detailed analysis, we store and compare the actual click-points.

Once the user had a chance to practice a few passwords, the main part of the experiment began. For each image, the user was asked to create a click-based graphical password that they could remember but that others will not be able to guess, and to pretend that it is protecting their bank information. After initial creation, the user was asked to confirm their password to ensure they could repeat their click-points. On successful confirma-

tion, the user was given 3D mental rotation tasks [33] as a distractor for at least 30 seconds. This distractor was presented to remove the password from their visual working memory, and thus simulate the effect of the passage of time. After this period of memory tasks, the user was provided the image again and asked to log in using their previously selected password. If the user could not confirm after two failed attempts or log in after one failed attempt, they were permitted to reset their password for that image and try again. If the user did not like the image and felt they could not create and remember a password on it, they were permitted to skip the image. Only two images had a significant number of skips: *paperclips* and *bee*. This suggests some passwords for these images were not repeatable, and we suspect our results for these images would show lower relative security in practice.

To avoid any dependence on the order of images presented, each user was presented a random (but unique) shuffled ordering of the 17 images used. Since most users did not make it through all 17 images, the number of graphical passwords created per image ranged from 32 to 40, for the 43 users. Two users had a “jumpy” mouse, but we do not expect this to affect our present focus – the location of selected click-points. This short-term study was intended to collect data on initial user choice; although the mental rotation tasks work to remove the password from working memory, it does not account for any effect caused by password resets over time due to forgotten passwords. The long-term study (Section 4) does account for this effect, and we compare the results.

#### 3.1 Results on Hot-Spots and Popular Clusters Observed

To explore the occurrence of hot-spotting in our lab user study, we assigned all the user click-points observed in the study to clusters as follows. Let  $R$  be the raw (unprocessed) set of click-points,  $M$  a list of temporary clusters, and  $V$  the final resulting set of clusters.

1. For each  $c_k \in R$ , let  $B_k$  be a temporary cluster containing click-point  $c_k$ . Temporarily assign all user click-points in  $R$  within  $c_k$ ’s T-region to  $B_k$ . Add  $B_k$  to  $M$ .
2. Sort all clusters in  $M$  by size, in decreasing order.
3. Greedily make permanent assignments of click-points to clusters as follows. Let  $B_\ell$  be the largest cluster in  $M$ . Permanently assign each click-point  $c_k \in B_\ell$  to  $B_\ell$ , then delete each  $c_k \in B_\ell$  from all other clusters in  $M$ . Delete  $B_\ell$  from  $M$ , and add  $B_\ell$  to  $V$ . Repeat until  $M$  is empty.

This process determines a set  $V$  of (non-empty) clusters and their sizes. We then calculate the observed

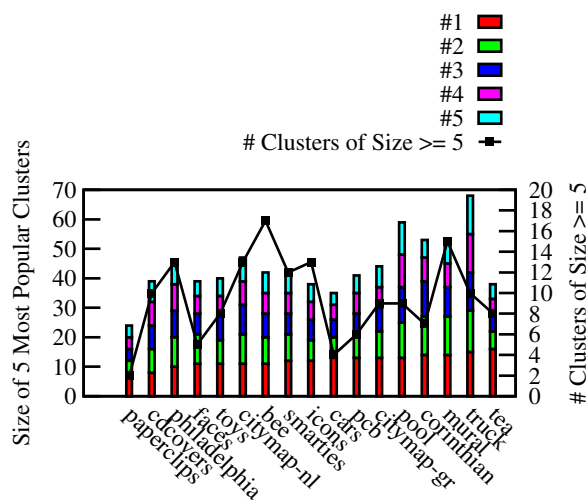


Figure 1: The five most popular clusters (in terms of size, i.e., # of times selected), and # of popular clusters ( $size \geq 5$ ). Results are from 32-40 users, depending on the image, for the final passwords created on each image. For *pcb*, which shows only 6 clusters of size  $\geq 5$ , the size of clusters 2-5 become 5, 5, 4, and 3 when counting at most one click from each user.

“probability”  $p_j$  (based on our user data set) of the cluster  $j$  being clicked, as cluster size divided by total clicks observed. When the probability  $p_j$  of a certain cluster is sufficiently high, we can place a confidence interval around it for future populations (of users who are similar in background to those in our study) using (1) as discussed below.

Each probability  $p_j$  estimates the probability of a cluster being clicked for a *single* click. For 5-click passwords, we approximate the probability that a user chooses cluster  $j$  in a password by  $P_j = 5 \times p_j$ . Note that the probability for a cluster  $j$  increases slightly as other clicks occur (due to the constraint of 5 distinct clusters in a password); we ignore this in our present estimate of  $P_j$ .

Our results in Figure 1 indicate a significant number of hot-spots for our sample of the full population (32 – 40 users per image). Previous “conservative” assumptions [47] were that half of the available alphabet of T-regions would be used in practice – or 207 in our case. If this were the case, and all T-regions in the alphabet were equi-probable, we would expect to see some clusters of size 2, but none of size 3 after 40 participants; we observed significantly more on all 17 images. Figure 1 shows that some images were clearly worse than others. There were many clusters of size at least 5, and some as large as 16 (see *tea* image). If a cluster in our lab study received 5 or more clicks – in which case we call it a *popular* or *high-probability* cluster – then statistically,

this allows determination of a confidence interval, using Equation (1) which provides the  $100(1 - \alpha)\%$  confidence interval for a population proportion [9, page 288].

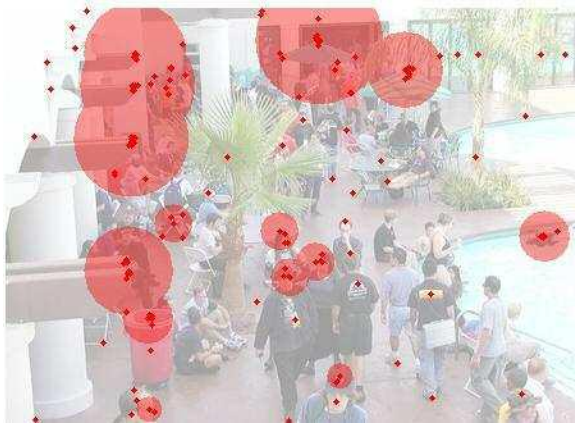
$$p \pm z_{\alpha/2} \sqrt{\frac{pq}{n}} \quad (1)$$

Here  $n$  is the total number of clicks (i.e., five times the number of users),  $p$  takes the role of  $p_j$ ,  $q = 1 - p$ , and  $z_{\alpha/2}$  is from a z-table. A confidence interval can be placed around  $p_j$  (and thus  $P_j$ ) using (1) when  $np \geq 5$  and  $nq \geq 5$ . For clusters of size  $k \geq 5$ ,  $p = \frac{k}{n}$ , then  $np = k$  and  $nq = n - k$ . In our case,  $n \geq 32 \cdot 5$  and  $n - k \geq 5$ , as statistically required to use (1).

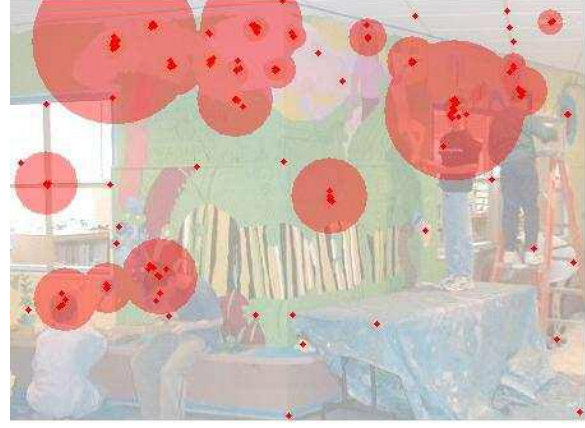
Table 1 shows these confidence intervals for four images, predicting that in future similar populations many of these points would be clicked by between 10-50% of users, and some points would be clicked by 20-60% of users with 95% confidence ( $\alpha = .05$ ). For example, in Table 1(a), the first row shows the highest frequency cluster (of size 13); as our sample for this image was only 35 users, we observed 37.1% of our participants choosing this cluster. Using (1), between 17.7% and 56.6% of users from future populations are expected to choose this same cluster (with 95% confidence).

Figure 1 and Table 1 show the popularity of the hottest clusters; Figure 1’s line also shows the number of popular clusters. The clustering effect evident in Figures 1, 2, and Table 1 clearly establishes that hot-spots are very prominent on a wide range of images. We further pursue how these hot-spots impact the practical security of full 5-click passwords in Section 4.2. As a partial summary, our results suggest that many images have significantly more hot-spots than would be expected if all T-regions were equi-probable. The *paperclips*, *cars*, *faces*, and *tea* images are not as susceptible to hot-spotting as others (e.g., *mural*, *truck*, and *philadelphia*). For example, the *cars* image had only 4 clusters of size at least 5, and only one with frequency at least 10. The *mural* image had 15 clusters of size at least 5, and 3 of the top 5 frequency clusters had frequency at least 10. Given our sample size for the *mural* image was only 36 users, these clusters are quite popular. This demonstrates the range of effect the background image can have (for the images studied).

Although previous work [46] suggests using intuition for choosing more secure background images (no further detail was provided), our results apparently show that intuition is not a good indicator. Of the four images used in other click-based graphical passwords studies, three showed a large degree of clustering (*pool*, *mural*, and *philadelphia*). Furthermore, two other images that we “intuitively” believed would be more secure background images were among the worst (*truck* and *citymap-nl*). The *truck* image had 10 clusters of size at least 5, and the top 5 clusters had frequency at least 13. Finding reliable automated predictors of more secure background images



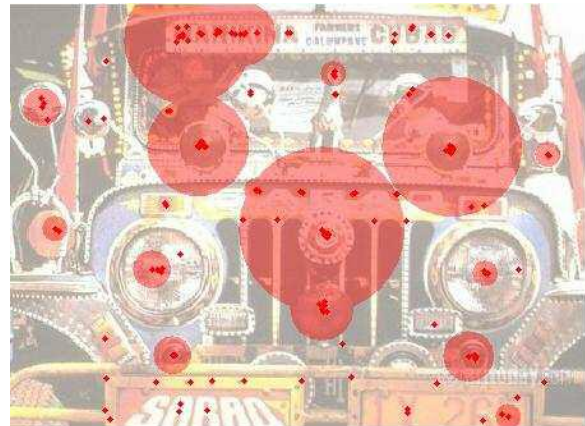
(a) *pool* (originally from [46, 47]; see Appendix A).



(b) *mural* (originally from [46]; see Appendix A).



(c) *philadelphia* (originally from [46]; see Figure 5).



(d) *truck* (originally from [12]).

Figure 2: Observed click-points. Halo diameters are 10 times the size of the underlying cluster, illustrating its popularity.

(a) <i>pool</i> image			(b) <i>mural</i> image		
Cluster size	$P_j$	95% CI ( $P_j$ )	Cluster size	$P_j$	95% CI ( $P_j$ )
13	0.371	(0.177; 0.566)	14	0.400	(0.199; 0.601)
12	0.343	(0.156; 0.530)	13	0.371	(0.177; 0.566)
12	0.343	(0.156; 0.530)	10	0.286	(0.114; 0.458)
11	0.314	(0.134; 0.494)	8	0.229	(0.074; 0.383)
11	0.314	(0.134; 0.494)	7	0.200	(0.055; 0.345)

(c) <i>philadelphia</i> image			(d) <i>truck</i> image		
Cluster size	$P_j$	95% CI ( $P_j$ )	Cluster size	$P_j$	95% CI ( $P_j$ )
10	0.286	(0.114; 0.458)	15	0.429	(0.221; 0.636)
10	0.286	(0.114; 0.458)	14	0.400	(0.199; 0.601)
9	0.257	(0.094; 0.421)	13	0.371	(0.177; 0.566)
9	0.257	(0.094; 0.421)	13	0.371	(0.177; 0.566)
7	0.200	(0.055; 0.345)	13	0.371	(0.177; 0.566)

Table 1: 95% confidence intervals for the top 5 clusters found in each of four images. The confidence intervals are for the percentage of users expected to choose this cluster in future populations.

remains an open problem. Our preliminary work with simple measures (image segmentation, corner detection, and image contrast measurement) does not appear to offer reliable indicators. Thus, we next explore the impact of hot-spotting across images to help choose two images for further analysis.

### 3.2 Measurement and Comparison of Hot-Spotting for Different Images

To compare the relative impact of hot-spotting on each image studied, we calculated two formal measures of password security for each image: entropy  $H(X)$ , in equation (2), and in equation (3), the expected number of guesses  $E(f(X))$  to correctly guess a password assuming the attacker knows the probabilities  $w_i > 0$  for each password  $i$ . The relationship between  $H(X)$  and  $E(f(X))$  for password guessing is discussed by Massey [26]. Of course in general, the  $w_i$  are unknown, and our study gives only very coarse estimates; nonetheless, we find it helpful to use this to develop an estimate of which images will have the least impact from hot-spotting. For (2) and (3),  $n$  is the number of passwords (of probability  $> 0$ ), random variable  $X$  ranges over the passwords, and  $w_i = \text{Prob}(X = x_i)$  is calculated as described below.

$$H(X) = - \sum_{i=1}^n w_i \cdot \log(w_i) \quad (2)$$

$$E(f(X)) = \sum_{i=1}^n i \cdot w_i, \text{ where } w_i \geq w_{i+1}, \text{ and } \quad (3)$$

$f(X)$  is the number of guesses before success. We calculate these measures based on our observed user data. For this purpose, we assume that users will choose from a set of click-points (following the associated probabilities), and combine 5 of them randomly. This assumption almost certainly over-estimates both  $E(f(X))$  and  $H(X)$  relative to actual practice, as it does not consider click-order patterns or dependencies. Thus, popular clusters likely reduce security by more than we estimate here.

We define  $C^V$  to be the set of all 5-permutations derivable from the clusters observed in our user study (as computed in Section 3.1). Using the probabilities  $p_j$  of each cluster, the probabilities  $w_i$  of each password in  $C^V$  are computed as follows. Pick a combination of 5 observed clusters  $j_1, \dots, j_5$  with respective probabilities  $p_{j_1}, \dots, p_{j_5}$ . For each permutation of these clusters, calculate the probability of that permutation occurring as a password. Due to our instructions that no two click-points in a password can fall in the same T-region, these probabilities change as each point is clicked. Thus, for password  $i = (j_1, j_2, j_3, j_4, j_5)$ ,  $w_i = p_{j_1} \cdot [p_{j_2} / (1 - p_{j_1})] \cdot [p_{j_3} / ((1 - p_{j_1}) \cdot (1 - p_{j_2}))] \cdot \dots$ .

The resulting set  $C^V$  is a set of click-based graphical passwords (with associated probabilities) that coarsely

approximates the effective password space if the clusters observed in our user study are representative of those in larger similar populations. We can order the elements of  $C^V$  using the probabilities  $w_i$  based on our user study. An ordered  $C^V$  could be used as the basis of an attack dictionary; this ordering could be much improved, for example, by exploiting expected patterns in click-order. See Section 4.2 for more details.

For comparison to previous “conservative” estimates that simply half of the available click-points (our T-regions) would be used in practice [47], we calculate  $C^U$ . We compare to  $C^U$  as it is a baseline that approximates what we would expect to see after running 32 users (the lowest number of users we have for any image), if previous estimates were accurate, and T-regions were equiprobable.  $C^U$  is the set of all permutations of clusters we expect to find after observing 32 users, assuming a uniformly random alphabet of size 207.

Fig. 3 depicts the entropy and expected number of guesses for  $C^V$ . Notice the range between images, and the drop in  $E(f(X))$  from  $C^U$  to values of  $C^V$ . Comparison to the marked  $C^U$  values for (1)  $H(X)$  and (2)  $E(f(X))$  indicates that previous rough estimates are a security overestimate for practical security in all images, some much more so than others. This is at least partially due to click-points not being equiprobable in practice (as illustrated by hot-spots), and apparently also due to the previously suggested effective alphabet size (half of the full alphabet) being an overestimate. Indeed, a large alphabet is precisely the theoretical security advantage that these graphical passwords have over text passwords. If the effective alphabet size is not as large as previously expected, or is not well-distributed, then we should reduce our expectations of the security.

These results appear to provide fair approximation of the entropy and expected number of guesses for the larger set of users in the field study; we performed these same calculations again using the field study data. For both of the two images, the entropy measures were within one bit of values measured here (less than a bit higher for *pool*, and about one bit lower for *cars*). The number of expected guesses increased for both images (by 1.3 bits for *cars*, and 2.5 bits for *pool*).

The variation across all images shows how much of an impact the background image can be, even when using images that are “intuitively” good. For example, the image that showed the most impact from hot-spotting was the *mural* image, chosen for an earlier PassPoints usability study [46]. We note that the *paperclips* image scores best in the charted security measures (its  $H(X)$  measure is within a standard deviation of  $C^U$ ); however, 8 of 36 users who created a password on this image could not perform the subsequent login (and skipped it – as noted earlier), so the data for this image represents some

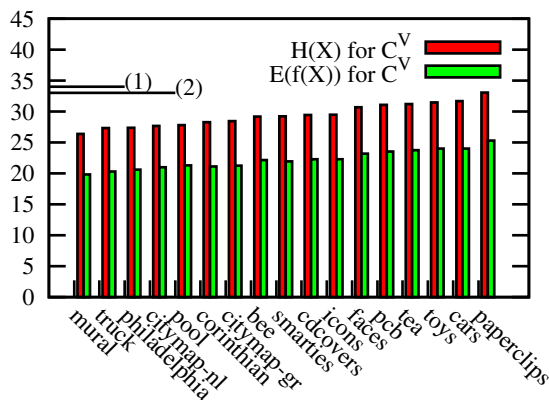


Figure 3: Security measures for each image (in bits).  $C^V$  is based on data from lab user study of 32–40 passwords (depending on image). For comparison to a uniform distribution, (1) marks  $H(X)$  for  $C^U$ , and (2) marks  $E(f(X))$  for  $C^U$ .

passwords that are not repeatable, and thus we suspect it would have lower relative security in practice.

Overall, one can conclude that image choice can have a significant impact on the resulting security, and that developing reliable methods to filter out images that are the most susceptible to hot-spotting would be an interesting avenue for future research.

We used these formal measures to make an informed decision on which images to use for our field study. Our goal was to give the PassPoints scheme the best chance (in terms of anticipated security) we could, by using one image (*cars*) that showed the least amount of clustering (with the best user success in creating a password), and also using another that ranked in the middle (*pool*).

## 4 Field Study and Harvesting Attacks

Here we describe a 7-week or longer (depending on the user), university-approved field study of 223 user accounts on two different background images. We collected click-based graphical password data to evaluate the security of this style of graphical passwords against various attacks. As discussed, we use the entropy and expected guesses measures from our lab study to choose two images that would apparently offer different levels of security (although both are highly detailed): *pool* and *cars*. The *pool* image had a medium amount of clustering (cf. Fig. 3), while the *cars* image had nearly the least amount of clustering. Both images had a low number of skips in the lab study, indicating that they did not cause problems for users with password creation.

**EXPERIMENTAL DETAILS.** We implemented a web-based version of PassPoints, used by three first-year undergraduate classes: two were first year courses for com-

puter science students, while the third was a first year course for non-computer science students enrolled in a science degree. The students used the system for at least 7 weeks to gain access to their course notes, tutorials, and assignment solutions. For comparison with previous usability studies on the subject, and our lab study, we used an image size of  $451 \times 331$  pixels. After the user entered their username and course, the screen displayed their background image and a small black square above the image to indicate their tolerance square size. For about half of users (for each image), a  $19 \times 19$  T-region was used, and for the other half, a  $13 \times 13$  T-region.<sup>2</sup> The system enforced that each password had to be 5 clicks and that no click-point could be within  $t = 9$  pixels of another (vertically and horizontally). To complete initial password creation, a user had to successfully confirm their password once. After initial creation, users were permitted to reset their password at any time using a previously set secret question and answer.

Users were permitted to login from any machine (home, school, or other), and were provided an online FAQ and help. The users were asked that they keep in mind that their click-points are a password, and that while they will need to pick points they can remember, not to pick points that someone else will be able to guess. Each class was also provided a brief overview of the system, explaining that their click-points in subsequent logins must be within the tolerance shown by a small square above the background image, and that the input order matters. We only use the final passwords created by each user that were demonstrated as successfully recalled at least one subsequent time (i.e., at least once after the initial create and confirm). We also only use data from 223 out of 378 accounts that we would consider, as this was the number that provided the required consent. These 223 user accounts map to 189 distinct users as 34 users in our study belonged to two classes; all but one of these users were assigned a different image for each account, and both accounts for a given user were set to have the same error tolerance. Of the 223 user accounts, 114 used *pool* and 109 used *cars* as a background image.

### 4.1 Field Study Hot Spots and Relation to Lab Study Results

Here we present the clustering results from the field study, and compare results to those on the same two images from the lab study. Fig. 4b shows that the areas that were emerging as hot-spots from the lab study (recall Fig. 2a) were also popular in the field study, but other clusters also began to emerge. Fig. 4b shows that even our “best” image from the lab study (in terms of apparent resistance to clustering) also exhibits a clustering effect after gathering 109 passwords. Table 2 provides a closer

examination of the clustering effect observed.

Image Name	Size of most popular clusters					# clusters of size $\geq 5$
	# 1	# 2	# 3	# 4	# 5	
<i>cars</i>	26	25	24	22	22	32
<i>pool</i>	35	30	30	27	27	28

Table 2: Most popular clusters (field study).

These values show that on *pool*, there were 5 points that 24-31% of users chose as part of their password. On *cars*, there were 5 points that 20-24% of users chose as part of their password. The clustering on the *cars* image indicates that even highly detailed images with many possible choices have hot spots. Indeed, we were surprised to see a set of points that were this popular, given the small amount of observed clustering on this image from our smaller lab study.

The prediction intervals calculated from our lab study (recall Section 3) provide reasonable predictions of what we observed in the field study. For *cars*, the prediction intervals for 3 out of the 4 popular clusters were correct. For *pool*, the prediction intervals for 8 out of the 9 popular clusters were correct. The anomalous cluster on *cars* was still quite popular (chosen by 12% of users), but the lower end of the lab study’s prediction interval for this cluster was 20%. The anomalous cluster on *pool* was also still quite popular (chosen by 18% of users), but the lower end of the lab study’s prediction interval for this cluster was 19%.

These clustering results (and their close relationship to the lab study’s results) indicate that the points chosen from the lab study should provide a reasonably close approximation of those chosen in the field. This motivates our attacks based on the click-points harvested from the lab study.

## 4.2 Harvesting Attacks: Method & Results

We hypothesized that due to the clustering effect we observed in the lab study, human-seeded attacks based on data harvested from other users might prove a successful attack strategy against click-based graphical passwords. Here we describe our method of creating these attacks, and our results are presented below.

Table 3 provides the results of applying various attack dictionaries based on our harvested data, and their success rates when applied to our field study’s password database.<sup>3</sup>

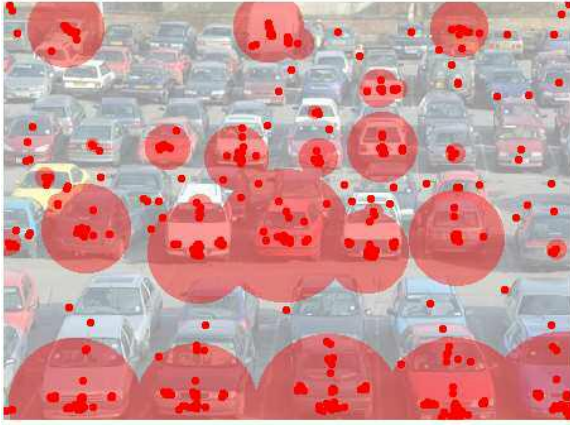
$C_u^R$  is a dictionary composed of all 5-permutations of click-points collected from  $u$  users. Note  $C_u^R$  bit-size is a slight overestimate, as there are some combinations of points that would not constitute a valid password, due to two or more points being within  $t = 9$  pixels of each other. If this were taken into account, our

attacks would be slightly better. In our lab study,  $u = 33$  for *cars*, and  $u = 35$  for *pool*. Thus, the size of  $C_u^R$  for *cars* is  $P(165, 5) = 2^{36.7}$  entries, and for *pool* is  $P(175, 5) = 2^{37.1}$  entries.  $C_u^V$  is a dictionary composed of all 5-permutations of the *clusters* calculated (using the method described in Section 3.1) from the click-points from  $u$  users. Thus, the alphabet size (and overall size) for  $C_u^V$  is smaller under the same number of users than in a corresponding  $C_u^R$  dictionary. Note that all of these dictionary sets can be computed on-the-fly from base data as necessary, and thus need not be stored.

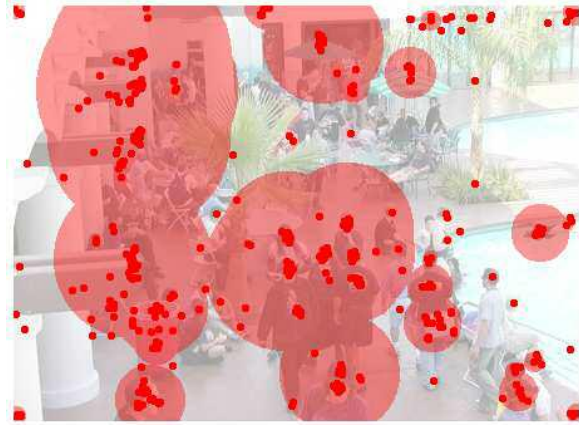
Table 3 illustrates the efficacy of seeding a dictionary with a small number of user’s click-points. The most striking result shown is that initial password choices harvested from 15 users, in a setting where long term recall is not required, can be used to generate (on average) 27% of user passwords for *pool* (see  $C_{15}^R$ ). As we expected, *cars* was not as easily attacked as *pool*; more user passwords are required to seed a dictionary that achieves similar success rates (see  $C_{25}^R$ ).

We also tried these attacks using a small set of field study user passwords to seed an attack against the remaining field study user passwords. The result, in Table 4, shows a difference between the lab study and the field study (final) passwords; however, there remains sufficient similarity between the two groups to launch effective attacks using the lab-harvested data. One possible reason for the differences in user choice between the two studies is that the field study users may not have been as motivated as the lab study users to create “difficult to guess” graphical passwords. It is unclear how a user might measure whether they are creating a graphical password that is difficult to guess, and whether in trying, if users would actually change their password’s strength; one study [36] shows that only 40% of users actually change the complexity of their text passwords according to the security of the site. Another equally possible explanation might be that the lab study users chose more difficult passwords than they would have in practice, as they were aware there was no requirement for long term recall, and also did not have a chance to forget and subsequently reset their passwords to something more memorable. With our current data, it is not clear whether we can conclusively determine a reason for these differences.

Next we examined the effect of click-order patterns as one method to capture a user’s association between points, and reduce our dictionary sizes. For each image, we select one dictionary to optimize with click-order patterns. This dictionary is one of the ten randomly selected  $C^V$  subsets that were averaged (results of this average are in Table 3). We selected the dictionary whose guessing success was closest to the average reported in Table 3. The success rate that these dictionaries achieve (be-



(a) *cars* (originally from [5]).



(b) *pool* (originally from [46, 47]).

Figure 4: Observed clustering (field study). Halo diameter is  $5 \times$  the number of underlying clicks.

Set	<i>cars</i> ( $u = 33$ )					<i>pool</i> ( $u = 35$ )				
	$m$	bitsize	# passwords guessed out of 109			$m$	bitsize	# passwords guessed out of 114		
			avg	min	max			avg	min	max
$C_u^R$	165	36.7	37(34%)	†	†	175	37.1	59(52%)	†	†
$C_u^V$	104	33.4	22(20%)	†	†	77	31.1	41(36%)	†	†
$C_{25}^R$	125	34.7	24(22%)	9(8%)	35(32%)	125	34.7	42(37%)	29(25%)	56(49%)
$C_{25}^V$	85	31.9	21(19%)	7(6%)	27(25%)	59	29.2	34(29%)	19(17%)	47(41%)
$C_{20}^R$	100	33.1	22(20%)	8(7%)	32(29%)	100	33.1	35(31%)	24(21%)	55(48%)
$C_{20}^V$	72	30.6	17(16%)	8(7%)	30(28%)	52	28.2	28(25%)	18(16%)	43(38%)
$C_{15}^R$	75	30.9	14(13%)	4(4%)	25(23%)	75	30.9	30(27%)	20(18%)	45(39%)
$C_{15}^V$	56	28.8	12(11%)	4(4%)	24(22%)	41	26.4	26(23%)	14(12%)	43(38%)

Table 3: Dictionary attacks using different sets. All subsets of users (after the first two rows) are the result of 10 randomly selected subsets of  $u$  short-term study user passwords. For rows 1 and 2, note that  $u = 33$  and 35.  $m$  is the alphabet size, which defines the dictionary bitsize. See text for descriptions of  $C^V$  and  $C^R$ . †The first two rows use all data from the short-term study to seed a single dictionary, and as such, there are no average, max, or min values to report.

fore applying click-order patterns) is provided in the first row of Table 5.

We hypothesized that many users will choose passwords in one (or a combination) of six simple click-order patterns: right to left (RL), left to right (LR), top to bottom (TB), bottom to top (BT), clockwise (CW), and counter-clockwise (CCW). Diagonal (DIAG) is a combination of a consistent vertical and horizontal direction (e.g., both LR and TB). Note that straight lines also fall into this category; for example, when  $(x_i, y_i)$  is a horizontal and vertical pixel coordinate, the rule for LR is  $(x_1 \leq x_2 \leq x_3 \leq x_4 \leq x_5)$ , so a vertical line of points would satisfy this constraint. We apply our base attack dictionaries (one for each image), under various sets of these click-order pattern constraints to determine their success rates and dictionary sizes. This method only initiates the exploration of other ways that click-

based graphical passwords could be analyzed for patterns in user choice. We expect this general direction will yield other results, including patterns due to mnemonic strategies (e.g., clicking all red objects).

The results shown in Table 5 indicate that, on average for the *pool* image, using only the diagonal constraint will reduce the dictionary size to 16 bits, while still cracking 12% of passwords. Similarly, for the *cars* image, using only this constraint will reduce the dictionary to 18 bits, while still cracking 10% of passwords. The success rate of our human-seeded attack is comparable to recent results on cracking text-based passwords [23], where 6% of passwords were cracked with a 1.2 million entry dictionary (almost 2 bits larger than our DIAG dictionary based on harvested points of 15 users for *cars*, and 4 bits larger for DIAG based on 15 users for *pool*). Furthermore, unlike most text dictionaries, we

Dictionary	<i>cars</i>			<i>pool</i>		
	$m$	bitsize	# passwords guessed	$m$	bitsize	# passwords guessed
$C_{20, longterm}^R$	100	33.1	29/89 (33%)	100	33.1	52/94 (55%)
$C_{10, longterm}^R$	50	27.9	23/99 (23%)	50	27.9	22/104 (21%)

Table 4: Dictionary attack results, using the first 20 and 10 users from the long term study to seed an attack against the others.  $m$  is the alphabet size. See text for descriptions of  $C^V$  and  $C^R$ .

Click-order pattern	<i>cars</i> image		<i>pool</i> image	
	# passwords guessed of 109	dictionary size (bits)	# passwords guessed of 114	dictionary size (bits)
$C_{15}^V$ (with no pattern)	13 (12%)	29.2	22 (19%)	27.1
LR, RL, CW, CCW, TB, BT	12 (11%)	25.6	22 (19%)	23.4
LR, RL	11 (10%)	23.8	19 (17%)	22.0
TB, BT	12 (11%)	24.4	15 (13%)	21.9
CW, CCW	0 (0%)	24.0	4 (4%)	21.7
DIAG	11 (10%)	18.4	14 (12%)	16.2

Table 5: Effect of incorporating click-order patterns on dictionary size and success, as applied to a representative dictionary of clusters gathered from 15 users. Results indicate that the DIAG pattern produces the smallest dictionary, and still guesses a relatively large number of passwords.

do not need to store the entire dictionary as it is generated on-the-fly from the alphabet. At best, this indicates that these graphical passwords are slightly less secure than the text-based passwords they have been proposed to replace. However, the reality is likely worse. The analogy to our attack is collecting text passwords from 15 users, and generating a dictionary based on all permutations of the characters harvested, and finding it generated a successful attack. The reason most text password dictionaries succeed is due to known dependent patterns in language (e.g., using di or tri-grams in a Markov model [29]). The obvious analogy to this method has not been yet attempted, but would be another method of further reducing the dictionary size.

## 5 Purely Automated Attacks Using Image Processing Tools

Here we investigate the feasibility of creating an attack dictionary for click-based graphical passwords by purely automated means. Pure automation would sidestep the need for human-seeding (in the form of harvesting points), and thus should be easier for an attacker to launch than the attacks presented in Section 4. We create this attack dictionary by modelling user choice using a set of image processing methods and tools. The idea is that these methods may help predict hot-spots by automated means, leading to more efficient search orderings for exhaustive attacks. This could be used for modeling

attackers constructing attack dictionaries, and proactive password checking.

### 5.1 Identifying Candidate Click-Points

We begin by identifying details of the user task in creating a click-based graphical password. The user must choose a set of points (in a specific order) that can be remembered in the future. We do not focus on mnemonic strategies for these automated dictionaries (although they could likely be improved using the click-order patterns from Section 4.2), but rather the basic features of a point that define candidate click-points. To this end, we identify a *candidate click-point* to be a point which is: (1) *identifiable* with precision within the system’s error tolerance; and (2) *distinguishable* from its surroundings, i.e., easily picked out from the background. Regarding (1), as an example, the *pool* image has a red garbage can that is larger than the  $19 \times 19$  error tolerance; to choose the red garbage can, a user must pick a *specific* part of it that can be navigated to again (on a later occasion) with precision, such as the left handle. Regarding (2), as an example, it is much easier to find a white logo on a black hat than a brown logo on a green camouflage hat.

For modelling purposes, we hypothesize that the fewer candidate click-points (as defined above) that an image has, the easier it is to attack. We estimate candidate click-points by implementing a variation of Itti et al.’s bottom-up model of visual attention (VA) [17], and combining it with Harris corner detection [16].

Corner detection picks out the areas of an image that have variations of intensity in horizontal and vertical directions; thus we expect it should provide a reasonable measure of whether a point is identifiable. Itti et al.'s VA determines areas that stand out from their surroundings, and thus we expect it should provide a reasonable measure of a point's distinguishability. Briefly, VA calculates a saliency map of the image based on 3 channels (color, intensity, and orientation) over multiple scales. The saliency map is a grayscale image whose brighter areas (i.e., those with higher intensity values) represent more conspicuous locations. A viewer's focus of attention should theoretically move from the most conspicuous locations (represented by the highest intensity areas on the saliency map) to the least. We assume that users are more likely to choose click-points from areas which draw their visual attention.

We implemented a variation of VA and combined it with Harris corner detection to obtain a prioritized list of candidate click-points (*CCP-list*) as follows. (1) Calculate a VA saliency map (see Fig. 5(b)) using slightly smaller scales than Itti et al. [17] (to reflect our interest in smaller image details). The higher-intensity pixel values of the saliency map reflect the most "conspicuous" (and distinguishable) areas. (2) Calculate the corner locations using the Harris corner detection function as implemented by Kovessi [22]<sup>4</sup> (see Fig. 5(c)). (3) Use the corner locations as a bitmask for the saliency map, producing what we call a *cornered saliency map* (CSM). (4) Compute an ordered CCP-list of the highest to lowest intensity-valued CSM points. Similar to the focus-of-attention inhibitors used by Itti et al., we inhibit a CSM point (and its surrounding tolerance) once it has been added to the CCP-list so it is not chosen again (see Fig. 5(d)). The CCP-list is at least as long as the alphabet size (414), but is a prioritized list, ranking points from (the hypothesized) most to least likely.

## 5.2 Model Results

We evaluated the performance of the CCP-list as a model of user choice using the data from both the lab and field user studies. We first examined how well the first half (top 207) of the CCP-list overlaps with the observed high-probability clusters from our lab user study (i.e., those clusters of size at least 5). We found that this half-alphabet found all high-probability clusters on the *icons*, *faces*, and *cars* images, and most of the high-probability clusters on 11 of the 17 images. Most of the images that our model performed poorly on appeared to be due to the saliency map algorithm being overloaded with too much detail (*pcb*, *citymap-gr*, *paperclips*, *smarties*, and *truck* images). The other image on which this approach did not perform well (*mural*) appears to be due to the cor-

ner masking in step (3); the high probability points were centroids of circles.

To evaluate how well the CCP-list works at modelling users' *entire* passwords (rather than just a subset of click-points within a password), we used the top ranked one-third of the CCP-list values (i.e., the top 138 points for each image) to build a graphical dictionary and carry out a dictionary attack against the observed passwords from both user studies (i.e., on all 17 images in the lab study, and the *cars* and *pool* images again in the field study). We found that for some images, this 35-bit dictionary was able to guess a large number of user passwords (30% for the *icons* image and 29% for the *philadelphia* map image). For both short and long-term studies, our tool guessed 9.1% of passwords for the *cars* image. A 28-bit computer-generated dictionary (built from the top 51 ranked CCP-list alphabet) correctly guessed 8 passwords (22%) from the *icons* image and 6 passwords (17%) from the *philadelphia* image. Results of this automated graphical dictionary attack are summarized in Table 6.

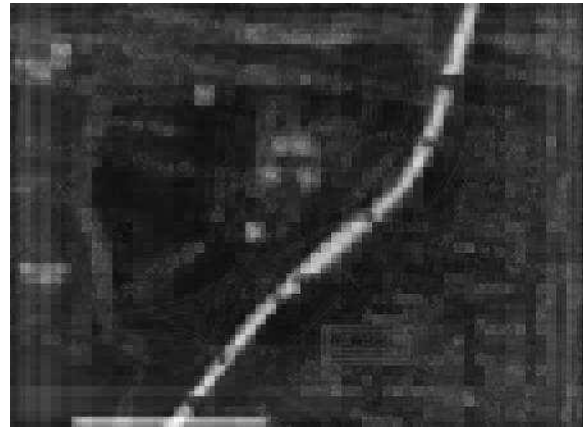
Image	passwords guessed (lab study)	passwords guessed (field study)
1. <i>paperclips</i>	2/36 (5.5%)	—
2. <i>cdcovers</i>	2/35 (5.7%)	—
3. <i>philadelphia</i>	10/35 (28.6%)	—
4. <i>toys</i>	2/39 (5.1%)	—
5. <i>bee</i>	1/40 (2.5%)	—
6. <i>faces</i>	0/32 (0.0%)	—
7. <i>citymap-nl</i>	1/34 (2.9%)	—
8. <i>icons</i>	11/37 (29.7%)	—
9. <i>smarties</i>	5/37 (13.5%)	—
10. <i>cars</i>	3/33 (9.1%)	10/109 (9.1%)
11. <i>pcb</i>	3/36 (8.3%)	—
12. <i>citymap-gr</i>	0/34 (0.0%)	—
13. <i>pool</i>	1/35 (2.9%)	2/114 (0.9%)
14. <i>mural</i>	1/36 (2.8%)	—
15. <i>corinthian</i>	3/35 (8.6%)	—
16. <i>truck</i>	1/35 (2.9%)	—
17. <i>tea</i>	2/38 (5.3%)	—

Table 6: Passwords correctly guessed (using a 35-bit dictionary based on a CCP-list). The number of target passwords is different for most images (32 to 40 for the lab study).

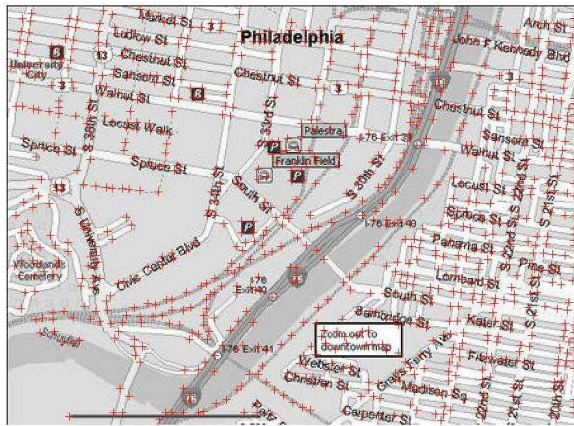
Figure 6 shows that the CCP-list does a good job of modelling observed user choices for some images, but not all images. This implies that on some images, an attacker performing an automated attack is likely to be able to significantly cut down his search space. This method also seems to perform well on the images for which the visual attention model made more definite decisions – the



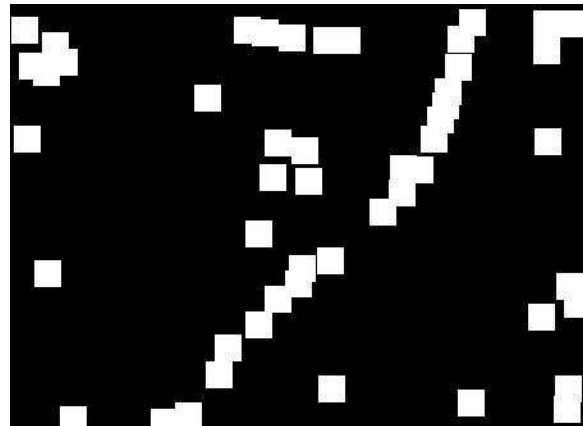
(a) Original image [46].



(b) Saliency map.



(c) Corner detection output.



(d) Cornered saliency map (CSM) after top 51 CCP-list points have been inhibited.

Figure 5: Illustration of our method of creating a CCP-list (best viewed electronically).

saliency map shows a smaller number of areas standing out, as indicated visually by a generally darker saliency map with a few high-intensity (white) areas. An attacker interested in any one of a set of accounts could go after accounts using a background image that the visual attention model performed well on.

In essence, this method achieves a reduction (by leaving out some “unlikely” points) from a 43-bit full password space to a 35-bit dictionary. The 43-bit full password space is the proper base for comparison here, since an actual attacker with no a priori knowledge must consider all T-regions in an image. However, we believe this model of candidate click-points could be improved through a few methods. The images that the model performed poorly on appeared to be due to failure in creating a useful visual attention model saliency map. The saliency maps seem to fail when there are no areas that stand out from their surroundings in the channels used in saliency map construction (color, intensity, and orientation). Further, centroids of objects that “stand out” to a

user will not be included in this model (as only corners are included); adding object centroids to the bitmask is thus an avenue for improvement.

## 6 Related Work

In the absence of password rules, practical text password security is understood to be weak due to common patterns in user choice. In a dated but still often cited study, Klein [21] determined a dictionary of 3 million words (less than 1 billionth of the entire 8-character password space) correctly guessed over 25% of passwords. Automated password cracking tools and dictionaries that exploit common patterns in user choice include *Crack* [28] and *John the Ripper* [30]. More recently, Kuo et al. [23] found John the Ripper’s English dictionary of 1.2 million words correctly guessed 6% of user passwords, and an additional 5% by also including simple permutations. In response to this well-known threat, methods to create less predictable passwords have emerged. Yan [48]

explores the use of passphrases to avoid password dictionary attacks. Jeyaraman et al. [20] suggest basing a passphrase upon an automated newspaper headline. In theory, creating passwords using these techniques should leave passwords less vulnerable to automated password cracking dictionaries and tools, although Kuo et al. [23] show this may not be the case. Proactive password checking techniques (e.g., [38, 7, 2]) are commonly used to help prevent users from choosing weak passwords.

Many variations of graphical passwords are discussed in surveys by Suo et al. [39] and Monroe et al. [27]. We discuss two general categories of graphical passwords: recognition-based and recall-based. In the interest of brevity, we focus on the areas closest to our work: click-based graphical passwords, and practical security analyses of user authentication methods.

Typical recognition-based graphical passwords require the user to recognize a set (or subset) of  $K$  previously memorized images. For example, the user is presented a set of  $N$  ( $> K$ ) images from which they must distinguish a subset of their  $K$  images. The user may be presented many panels of images before providing enough information to login. Examples are *Déjà Vu* [10], which uses random art images created by hash visualization [32]; *Passfaces* [35], whereby the set of images are all human faces; and *Story* [8], whereby the images are from various photo categories (e.g., everyday objects, locations, food, and people), with users encouraged to create a story as a mnemonic strategy. In the cognitive authentication scheme of Weinshall [44], a user computes a path through a grid of images based on the locations of those from  $K$ . The end of the path provides a number for the user to type, which was thought to protect the values of  $K$  from observers; Golle et al. [14] show otherwise.

Recall-based schemes can be further described as cued or uncued. An uncued scheme does not provide the user any information from which to create their graphical password; e.g., DAS (Draw-A-Secret) [19] asks users to draw a password on a background grid. Cued schemes show the user something that they can base their graphical password upon. A click-based password using a single background image is an example of a cued graphical password scheme where the user password is a sequence of clicks on a background image. Blonder [4] originally proposed the idea of a graphical password with a click-based scheme where the password is one or more clicks on predefined image regions. In the Picture Password variation by Jansen et al. [18], the entire image is overlaid by a visible grid; the user must click on the same grid squares on each login.

Birget et al. [3] allow clicking anywhere on an image with no visible grid, tolerating error through “robust discretization”. Wiedenbeck et al. [45, 46, 47] implement this method as PassPoints, and study its usability includ-

ing: memorability, general perception, error rates, the effect of allowed error tolerance, the effect of image choice on usability, and login and creation times. They report the usability of PassPoints to be comparable to text passwords in most respects; the notable exception is a longer time for successful login. The implementation we study herein is also reported to have acceptable success rates, accuracy, and entry times [6].

Regarding explorations of the effect of user choice, Davis et al. [8] examine this in a variation of Passfaces and Story (see above), two recognition-based schemes which essentially involve choosing an image from one or more panels of many different images. Their user study found very strong patterns in user choice, e.g., the tendency to select images of attractive people, and those of the same racial background. The high-level idea of finding and exploiting patterns in user choice also motivated our current work, although these earlier results do not appear directly extendable to (cued recall) click-based schemes that select unrestricted areas from a single background image. Thorpe et al. [41, 42] discussed likely patterns in user choice for DAS (mirror symmetry and small stroke count), later corroborated through Tao’s user study [40]. These results also do not appear to directly extend to our present work, aside from the common general idea of attack dictionaries.

Lopresti et al. [24] introduce the concept of generative attacks to behavioral biometrics. Ballard et al. [1] generate and successfully apply a generative handwriting-recognition attack based on population statistics of handwriting, collected from a random sample of 15 users with the same writing style. In arguably the most realistic study to date of the threats faced by behavioral biometrics, they found their generative attacks to be more effective than attacks by skilled and motivated forgers [1]. Our most successful attack from Section 4.2 may also be viewed as generative in nature; it uses click-points harvested from a small population of users from another context (the lab study), performs some additional processing (clustering), and recombines subsets of them as guesses. Our work differs in its application (click-based graphical passwords), and in the required processing to generate a login attempt.

## 7 Discussion and Concluding Remarks

Our results demonstrate that harvesting data from a small number of human users allows quite effective offline guessing attacks against click-based graphical passwords. This makes individual users vulnerable to targeted (spear) attacks, as one should assume that an attacker could find out the background image associated with a target victim, and easily gather a small set of human-generated data for that image by any number of

means. For instance, an attacker could collect points by protecting an attractive web service or contest site with a graphical password. Alternatively, an attacker could pay a small group of people or use friends. This at least partially defeats the hope to improve one's security in a click-based scheme through a customized image.

We found that our human-seeded attack strategy was quite successful, guessing 36% of passwords with a 31-bit dictionary in one instance, and 20% of passwords with a 33-bit dictionary in another. Preliminary work shows that click-order patterns can be used to further reduce the size of these dictionaries, while maintaining similar success rates. The success of our human-seeded attack dictionaries appears to be related to the amount of hot-spotting on an image. The prevalence and impact of hot-spots contrasts earlier views which underplayed their potential impact, and suggestions [47] that any highly detailed image may be a good candidate. Our studies allow us to update previous assumptions that half of all click-regions on an image will be chosen by users. After collecting 570 and 545 points, we only observed 111 and 133 click-regions (for *pool* and *cars* respectively); thus, one quarter to one third of all click-regions would be a more reasonable estimate even from highly detailed images, and the relative probabilities of these regions should be expected to vary quite considerably.

Our purely automated attack using a combination of image processing measures (which likely can be considerably improved) already gives cause for concern. For images on which Itti et al.'s [17] visual attention model worked well, our model appeared to do a reasonable job of predicting user choice. For example, an automatically-generated 28-bit dictionary from our tools guessed 8 out of 37 (22%) observed passwords for the *icons* image, and 6 out of 35 (17%) for the *philadelphia* image. Our tools guessed 9.1% of passwords for the *cars* image in both the short-term lab and long-term field studies. Improvements to pursue include adding object centroids to the bitmask used in creating the cornered saliency map.

Our attack strategies (naturally) could be used defensively, as part of proactive password checking [38, 7, 2]. Thus, an interesting avenue for future work would be to determine whether graphical password users create other predictable patterns when their choices are disallowed by proactive checking. Additionally, the visual attention model may be used proactively to determine background images to avoid, as those images on which the visual attention model performed well (e.g., identifies some areas as much more interesting than others) appear more vulnerable to the purely automated attacks from Section 5.

An interesting remaining question is whether altering parameters (e.g., pixel sizes of images, tolerance settings, number of click-points) in an attempt to improve security can result in a system with acceptable security and us-

ability simultaneously. Any proposal with significantly varied parameters would require new user studies exploring hot-spotting and usability.

Overall, the degree of hot-spotting confirmed by our studies, and the successes of the various attack strategies herein, call into question the viability of click-based schemes like PassPoints in environments where off-line attacks are possible. Indeed in such environments, a 43-bit full password space is clearly insufficient to start with, so one would assume some tolerable level of password stretching (e.g., [15, 34]) would be implemented to increase the difficulty of attack. Regardless of these implementation details, click-based graphical password schemes may still be a suitable alternative for systems where offline attacks are not possible, e.g., systems currently using PIN numbers.

## Acknowledgments

We thank Sonia Chiasson and Robert Biddle for their cooperative effort with us in running the user studies. We are grateful to Prosenjit Bose, Louis D. Nel, Weixuan Li, and their Fall 2006 classes for participating in our field study. We also thank Anthony Whitehead for recommending relevant work on visual attention and image segmentation. The first author acknowledges NSERC for funding a Canada Graduate Scholarship. The second author acknowledges NSERC for funding a NSERC Discovery Grant and his Canada Research Chair in Network and Software Security. We thank Fabian Monrose, and the anonymous reviewers for their insightful suggestions for improving this paper.

## Notes

<sup>1</sup>Version: May 13, 2007. A preliminary version of this paper was available as a Technical Report [43].

<sup>2</sup>Analysis showed little difference between the points chosen for these different tolerance groups.

<sup>3</sup>A preliminary version [43] had a small technical error causing some numbers to be less than shown herein in Tables 3 and 5.

<sup>4</sup>As *harris(image, 1, 1000, 3)*

## References

- [1] L. Ballard, F. Monrose, and D. Lopresti. Biometric Authentication Revisited: Understanding the Impact of Wolves in Sheep's Clothing. In *15th Annual USENIX Security Symposium*, pages 29–41, 2006.
- [2] F. Bergadano, B. Crispo, and G. Ruffo. High Dictionary Compression for Proactive Password Checking. *ACM Trans. Inf. Syst. Secur.*, 1(1):3–25, 1998.

- [3] J.C. Birget, D. Hong, and N. Memon. Robust Discretization, with an Application to Graphical Passwords. *IEEE Transactions on Information Forensics and Security*, 1:395–399, 2006.
- [4] G. Blonder. Graphical Passwords. United States Patent 5,559,961, 1996.
- [5] Ian Britton. <http://www.freefoto.com>, accessed Feb. 2, 2007.
- [6] S. Chiasson, R. Biddle, and P.C. van Oorschot. A Second Look at the Usability of Click-based Graphical Passwords. In *Symposium on Usable Privacy and Security (SOUPS)*, 2007.
- [7] C. Davies and R. Ganesan. BApaswd: A New Proactive Password Checker. In *16th National Computer Security Conference*, pages 1–15, 1993.
- [8] D. Davis, F. Monroe, and M.K. Reiter. On User Choice in Graphical Password Schemes. In *13th USENIX Security Symposium*, 2004.
- [9] J.L. Devore. *Probability and Statistics for Engineering and the Sciences*. Brooks/Cole Publishing, Pacific Grove, CA, USA, 4th edition, 1995.
- [10] R. Dhamija and A. Perrig. Déjà Vu: A User Study Using Images for Authentication. In *9th USENIX Security Symposium*, 2000.
- [11] P.F. Felzenszwalb and D.P. Huttenlocher. Efficient Graph-Based Image Segmentation. *Int. J. Computer Vision*, 59(2), 2004. Code available from: <http://people.cs.uchicago.edu/pff/segment/>.
- [12] FreeImages.com. <http://www.freeimage.com>, accessed Feb. 2, 2007.
- [13] Freeimages.co.uk. <http://www.freeimage.co.uk>, accessed Feb. 2, 2007.
- [14] P. Golle and D. Wagner. Cryptanalysis of a Cognitive Authentication Scheme. *Cryptology ePrint Archive*, Report 2006/258, 2006. <http://eprint.iacr.org/>.
- [15] J. A. Halderman, B. Waters, and E. W. Felten. A Convenient Method for Securely Managing Passwords. In *Proceedings of the 14th International World Wide Web Conference*, pages 471–479. ACM Press, 2005.
- [16] C.G. Harris and M.J. Stephens. A Combined Corner and Edge Detector. In *Proceedings Fourth Alvey Vision Conference*, pages 147–151, 1988.
- [17] L. Itti, C. Koch, and E. Niebur. A Model of Saliency-Based Visual Attention for Rapid Scene Analysis. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 20(11):1254–1259, 1998.
- [18] W. Jansen, S. Gavrilla, V. Korolev, R. Ayers, and Swanstrom R. Picture Password: A Visual Login Technique for Mobile Devices. NIST Report: NISTIR 7030, 2003.
- [19] I. Jermyn, A. Mayer, F. Monroe, M. Reiter, and A. Rubin. The Design and Analysis of Graphical Passwords. In *8th USENIX Security Symposium*, 1999.
- [20] S. Jeyaraman and U. Topkara. Have the Cake and Eat it too - Infusing Usability into Text-Password Based Authentication Systems. In *21st ACSAC*, pages 473–482, 2005.
- [21] D. Klein. Foiling the Cracker: A Survey of, and Improvements to, Password Security. In *The 2nd USENIX Security Workshop*, pages 5–14, 1990.
- [22] P. D. Kovesi. MATLAB and Octave Functions for Computer Vision and Image Processing. Univ. Western Australia. Available from: <http://www.cse.uwa.edu.au/p/research/matlabfn/>.
- [23] C. Kuo, S. Romanosky, and L.F. Cranor. Human Selection of Mnemonic Phrase-based Passwords. In *2nd Symp. Usable Privacy and Security (SOUPS)*, pages 67–78, New York, NY, 2006. ACM Press.
- [24] Daniel P. Lopresti and Jarret D. Raim. The Effectiveness of Generative Attacks on an Online Handwriting Biometric. In *AVBPA*, pages 1090–1099, 2005.
- [25] S. Madigan. Picture Memory. In John C. Yuille, editor, *Imagery, Memory and Cognition*, pages 65–89. Lawrence Erlbaum Associates, N.J., U.S.A., 1983.
- [26] J.L. Massey. Guessing and Entropy. In *ISIT: Proceedings IEEE International Symposium on Information Theory*, page 204, 1994.
- [27] F. Monroe and M. K. Reiter. Graphical Passwords. In L. Cranor and S. Garfinkel, editors, *Security and Usability*, ch. 9, pages 147–164. O’Reilly, 2005.
- [28] A. Muffett. Crack password cracker, 2006. <http://www.crypticide.com/user/alecm/security/cfa.html>, accessed Nov. 9, 2006.
- [29] Arvind Narayanan and Vitaly Shmatikov. Fast Dictionary Attacks on Passwords Using Time-space Tradeoff. In *CCS ’05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 364–372, 2005.
- [30] Openwall Project. John the Ripper password cracker, 2006. <http://www.openwall.com/john/>, accessed Nov. 9, 2006.
- [31] Passlogix. <http://www.passlogix.com>, accessed Feb. 2, 2007.
- [32] A. Perrig and D. Song. Hash Visualization: A New Technique to Improve Real-World Security. In *International Workshop on Cryptographic Techniques and E-Commerce*, pages 131–138, 1999.
- [33] M. Peters, B. Laeng, K. Latham, M. Jackson, R. Zaiyouna, and C. Richardson. A Redrawn Vandenberg and Kuse Mental Rotations Test: Different Versions and Factors That Affect Performance. *Brain and Cognition*, 28:39–58, 1995.
- [34] N. Provos and D. Mazieres. A Future-Adaptable Password Scheme. In *Proceedings of the USENIX Annual Technical Conference*, 1999.

- [35] Real User Corporation. About Passfaces, 2006. <http://www.realuser.com/about/aboutpassface.htm>, accessed Nov. 9, 2006.
- [36] Shannon Riley. What Users Know and What They Actually Do. *Usability News*, 8(1), February 2006. <http://psychology.wichita.edu/url/abilitynew/aword.htm>, accessed March 10, 2007.
- [37] SFR IT-Engineering. The Grafical Login Solution For your Pocket PC - visKey. <http://www.frsoftware.de/cm/pocetpc/viskey/index.html>, accessed March 18, 2007.
- [38] E.H. Spafford. OPUS: Preventing Weak Password Choices. *Comput. Secur.*, 11(3):273–278, 1992.
- [39] X. Suo, Y. Zhu, and G.S. Owen. Graphical Passwords: A Survey. In *21st Annual Computer Security Applications Conference (ACSAC)*, 2005.
- [40] H. Tao. Pass-Go, a New Graphical Password Scheme. Master’s thesis, University of Ottawa, 2006.
- [41] J. Thorpe and P.C. van Oorschot. Graphical Dictionaries and the Memorable Space of Graphical Passwords. In *13th USENIX Security Symposium*, 2004.
- [42] J. Thorpe and P.C. van Oorschot. Towards Secure Design Choices for Implementing Graphical Passwords. In *20th Annual Computer Security Applications Conference (ACSAC 2004)*. IEEE, 2004.
- [43] J. Thorpe and P.C. van Oorschot. Human-Seeded Attacks and Exploiting Hot-Spots in Graphical Passwords. Technical Report TR-05-07, School of Computer Science, Carleton University, Feb. 20, 2007.
- [44] D. Weinshall. Cognitive Authentication Schemes Safe Against Spyware (short paper). In *IEEE Symp. on Security and Privacy*, pages 295–300, 2006.
- [45] S. Wiedenbeck, J. Waters, J.C. Birget, A. Brodskiy, and N. Memon. Authentication using graphical passwords: Basic results. In *Human-Computer Interaction International (HCII 2005)*, 2005.
- [46] S. Wiedenbeck, J. Waters, J.C. Birget, A. Brodskiy, and N. Memon. Authentication Using Graphical Passwords: Effects of Tolerance and Image Choice. In *Symp. Usable Priv. & Security (SOUPS)*, 2005.
- [47] S. Wiedenbeck, J. Waters, J.C. Birget, A. Brodskiy, and N. Memon. PassPoints: Design and Longitudinal Evaluation of a Graphical Password System. *International J. of Human-Computer Studies (Special Issue on HCI Research in Privacy and Security)*, 63:102–127, 2005.
- [48] J. Yan, A. Blackwell, R. Anderson, and A. Grant. Password Memorability and Security: Empirical Results. *IEEE Security and Privacy*, 2(5):25–31, 2004.



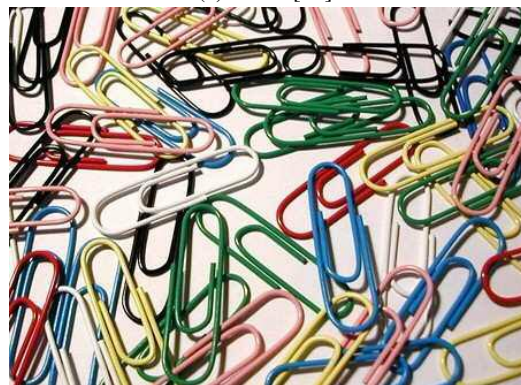
(a) cars [5].



(b) pool [46, 47].



(c) mural [46].



(d) paperclips [13].

## Appendix A - Subset of Images Used

# Halting Password Puzzles

## Hard-to-break Encryption from Human-memorable Keys

XAVIER BOYEN

*Voltage Security*

*xb@boyen.org*

### Abstract

We revisit the venerable question of “pure password”-based key derivation and encryption, and expose security weaknesses in current implementations that stem from structural flaws in Key Derivation Functions (KDF). We advocate a fresh redesign, named Halting KDF (HKDF), which we thoroughly motivate on these grounds:

1. By letting password owners choose the hash iteration count, we gain operational flexibility and eliminate the rapid obsolescence faced by many existing schemes.
2. By throwing a Halting-Problem wrench in the works of guessing that iteration count, we widen the security gap with any attacker to its theoretical optimum.
3. By parallelizing the key derivation, we let legitimate users exploit all the computational power they can muster, which in turn further raises the bar for attackers.

HKDFs are practical and universal: they work with any password, any hardware, and a minor change to the user interface. As a demonstration, we offer real-world implementations for the TrueCrypt and GnuPG packages, and discuss their security benefits in concrete terms.

## 1 Introduction

For a variety of reasons, it is becoming increasingly desirable for people leading an *electronic lifestyle* to attend to a last bastion of privacy: a stronghold defended by secret-key cryptography, and whose key exists only in its guardian’s mind. To this end, we study how “*pure password*”-based encryption can best withstand the most dedicated offline dictionary attacks—regardless of password strength.

### 1.1 Human-memorable Secrets

**Passwords.** Passwords in computer security are the purest form of secrets that can be kept in human memory,

independently of applications and infrastructures. They can be typed quickly and discreetly on a variety of devices, and remain effective in constrained environments with basic input and no output capabilities. Not surprisingly, passwords and passphrases have become the method of choice for human authentication and mental secret safekeeping, whether locally or remotely, in an on-line or offline setting.

Passwords have the added benefit to work on diminutive portable keypads that never leave the user’s control, guaranteeing that the secret will not be intercepted by a compromised terminal. User-owned and password-activated commercial devices include the *DigiPass* [12] for authorizing bank transactions, the *CryptoCard* [10] for generating access tokens, and the ubiquitous cellular phone which can be used for making payments via SMS over the GSM network.

Nevertheless, the widespread use of passwords for securing computer systems is often deplored by system administrators, due to their low entropy and a propensity to being forgotten unless written down, which in turn leads to onerous policies that users deem too difficult to follow [43]. In this work, by contrast, we seek not to change people’s habits in significant ways; rather, our goal is to maximize security for passwords that are actually used, no matter how weak these might be.

**Alternatives.** A number of alternatives have been suggested to alleviate the limitations of passwords, including inkblots [39], visual recognition [29], client-side puzzles [21, 11], interactive challenges [32], word labyrinths [6], but any of them has yet to gain much traction.

Multi-factor authentication systems seek not to replace passwords, but supplement them with a second or third form of authentication, which could be a physical token (e.g., SecurID [38]) or a biometric reading. These approaches are mostly effective in large organizations.

Compelling as these sophisticated proposals may be,

multi-factor authentication is no panacea, and the various mental alternatives to passwords tend to be slow, complex, and error-prone, and depend on a particular medium or infrastructure. For instance, mental puzzles typically require multiple rounds of interaction to gather enough entropy, and image recognition tasks will never work without a display. Simple portable keypads are pretty much out of the question. The usual criticisms that have been levelled at passwords, such as low entropy and poor cognitive retention, apply to these alternatives as well.

## 1.2 Application Contexts

**Online Uses.** In the online setting, the main use of passwords is for remote user authentication. Password-based Encrypted Key Exchange (EKE) [5] and Authenticated Key Exchange (PAKE) [17] protocols enable high-entropy session keys to be established between two or more parties that hold a low-entropy shared secret, ideally with mutual authentication. The threat model is the online attack, conducted by an opponent who can observe and corrupt the lines of communication, and sometimes also the transient state of a subset of the participants, but without access to the long-term storage where the password data are kept.

What makes the online setting favorable for password-based authentication, is that participants can detect (in zero knowledge) when an incorrect password is used, and terminate the protocol without leaking information. The attacker can always run a fresh instance of the protocol for every candidate password, but many EKE and PAKE protocols [20, 4, 8] achieve theoretically optimal security by ensuring that no adversary can do better than this. Online guessing is easy to detect in practice, and can be defeated by locking out accounts with repeated failures. Dealing with passwords in the pure online setting is in that respect a mostly solved problem, and is the topic of the ongoing IEEE 1363.2 standardization effort [19]. We will not discuss online passwords further.

**Offline Uses.** In the offline setting, passwords are mainly used for login and to encrypt data at rest in local storage. Typical applications of password-based encryption range from user-level encryption of PGP or S/MIME private keys, to kernel-level enforcement of access permissions, to hardware-level encryption of a laptop's hard disk by a security chip or by the drive itself.

Despite their limitations, passwords tend to be preferable to other types of credentials. Physical tokens able to store large cryptographic keys are susceptible to theft along with the laptop they are supposed to protect. Biometrics are inherently noisy and must trade security for reliability; they are also tied to a specific user and cannot

be revoked. Visual and other alternatives to passwords are often complex and too demanding for low-level operation or in embedded systems; at any rate they do not have clear security benefits over passwords.

The main threat faced by password-based encryption is the offline dictionary attack. Unlike the online guessing discussed earlier, in an offline attack the adversary has access to the complete ciphertext and all relevant information kept in storage—except the password—and does not need the cooperation of remote parties to carry out the attack. Tamper-resistant hardware may complicate ciphertext acquisition, but, past that point, the adversary is bound only by sheer computational power: this is what makes low-entropy passwords so much more damaging offline than online.

## 1.3 Password-based Encryption

Aside from the peril of dictionary attacks, passwords are not usable natively as encryption keys, because they are not properly distributed. Key Derivation Functions (KDF) let us solve this.

**Key Derivation.** The goal is to create a uniform and reproducible key from a password. The universally accepted practice is to mangle the password through a hash function a number of times, after blending it with random data called *salt* that is made public. The many hash iterations serve to make offline dictionary attacks slower, and the salt is to preclude using lookup tables as a shortcut [18, 30, 3]. Virtually all KDFs follow this model; however, it is not a panacea.

For ones, referring to the apparent futility of preventing (targeted) dictionary attacks, in the full version of their recent CRYPTO '06 paper, Canetti, Halevi, and Steiner [9] lament:

*[...] typical applications use a key-derivation-function such as SHA1 repeated a few thousand times to derive the key from the password, in the hope of slowing down off-line dictionary attacks. [...] Although helpful, this approach is limited, as it entails an eternal cat-and-mouse chase where the number of iterations of SHA1 continuously increases to match the increasing computing powers of potential attackers.*

Instead, these authors propose to treat the password as a path in a maze of CAPTCHAs [42], whose (secret) answers will provide the key. Alas, such augmented-password schemes tend to be unwieldy; here, gigabytes of CAPTCHAs must be pre-generated, and then retrieved in secret, which relegates it to local storage (lest an offline dictionary attack on the access pattern reveal the password).

In general, while it is true that secrets with visual or interactive components are likely to hamper mechanical enumeration, old-fashioned passwords will remain faster, less conspicuous, and much more convenient for humans to handle and recall. Still, the problem remains to design a good KDF.

**Iteration Count.** To perceive the difficulty of KDF design, recall that Unix' `crypt()` hashing for `/etc/passwd` back in the seventies took a quarter of a second [33] to perform two dozen iterations of the DES cipher (with salt). The original PKCS#5 key derivation standard from the early nineties [37] was content to use a “positive number” of applications of MD2 or MD5, but has since been updated [22] to recommend “at least 1000” iterations of MD5 or SHA1. This recommendation has been followed in the recent and well-regarded *TrueCrypt* software [40], albeit perhaps on the edge, with merely 2000 iterations of SHA1 or RIPEMD160, or 1000 iterations of WHIRLPOOL. Unfortunately, these numbers are set in stone in the *TrueCrypt* source code.

The custom “s2k” (string-to-key) function of *GnuPG* [15] is preset to hash a total of 65536 bytes based on the password, which amounts to a few thousand iterations of SHA1. Sadly, this number is once again hardcoded without user override. At least, the OPENPGP [7] format offers some flexibility in that regard, and *GnuPG* can be recompiled to hash up to a maximum of 65011712 bytes, without breaking compatibility with the official version. Still, even that ostensibly large number appears pathetic by today's standards, as it takes only two seconds to digest those 65 million bytes on a 1.5 GHz laptop *circa* 2005.

## 1.4 The Problem, and Our Solution

The balancing act in KDF design is to choose a large enough iteration count to frustrate a dictionary attack, but not so large as to inconvenience the user. Any choice made today is likely to prove wholly inadequate a few years from now. Furthermore, this assessment should be made in view of the lifespan and sensitivity of the plaintext, as well as the estimated strength of the password—two crucial tidbits of which only the actual user (and not the system designer) is privy.

**Security Maximization and User Programmability.** Given the constraints, the primary goal is to maximize—by technical means—the “gap” between user inconvenience and the costs inflicted on attackers. Secondly, it is crucial—for policy and deeper reasons—that users be free to vary the (secret) level of inconvenience they are willing to accept on a case-by-case basis. In essence, we:

- (i) let the user choose the amount of work he or she deems appropriate for the task,
- (ii) keep the choice secret from attackers (and allow the user to forget it too),
- (iii) and ensure that all user-side computing power can be exploited.

We emphasize again that human-selected passwords tend to be by far the weakest link in a typical cryptographic chain [26], which is why we seek to squeeze as much security from them as we can.

**“Halting” Key Derivation Functions.** HKDFs are the practical embodiment of all the above requirements. They consist of two algorithms, *Prepare* and *Extract*. The principle is as follows:

- To create a random encryption key, the user launches a randomized algorithm *HKDF.Prepare* on the password, lets it crunch for a while, and interrupts it manually using the user interface, to obtain an encryption key along with some public string to be stored with the ciphertext.
- To recover the same key subsequently, the user applies a deterministic algorithm *HKDF.Extract* on the password and the public string from the first phase. The algorithm halts spontaneously when it recognizes that it has recovered the correct key, barring which it can be reset manually.

Thus, if the user entered the correct password, *HKDF.Extract* will halt and output the correct key after roughly the same amount of time as the user had let *HKDF.Prepare* run in the setup phase. However, if the user entered a wrong password, at some point he or she will find that it is taking too long and will have the option to stop the process manually in order to try again.

Notice that the public string causes the derived key to be a randomized function of the password, and thus also plays the role of “salt”. HKDFs can be used as drop-in substitutes for regular KDFs, pending addition in the user interface of a button for interrupting the computation in progress.

**HKDF Ramifications.** The above idea is as simple as it is powerful, though surprisingly it has not been investigated or implemented before. Ramifications are deep, however:

1. (Stronger crypto) Two extra bits of security can be reclaimed *from any password*.

A paradoxical result that we prove in this paper is that, if the attacker does not know the iteration count, and is then compelled to use a “dovetail” search strategy with many restarts, then the attack

effort is multiplied by  $\sim 4\times$  (a 2-bit security gain), at no cost to the user.

Intuitively, our design will force any game-theoretic optimal brute-force attacker to overshoot the true iteration count when trying out wrong passwords. By contrast, when the user enters the correct password, the key derivation process will be halted as soon as the programmed number of iterations is reached (using some mechanism for detecting that this is the case).

2. (Flexible policies) Long-term memorable passwords for key recovery become a possibility.

Sophisticated users should be able to choose any password that they will remember in the long term, even with low entropy, as long as they are used with a large enough iteration count to keep brute-force attackers at bay (at the cost of slowing down legitimate uses correspondingly).

This opens the possibility of using multiple passwords of reciprocal strength and memorability: one high-entropy password with a small iteration count for fast everyday use; and a second, much more memorable password for the long term, protected by a very large iteration count, to be used as a backup if the primary password is forgotten.

3. (Future proofing) Password holders automatically keep pace with password crackers.

Indeed, if every time a user's password is changed, the iteration count is selected to take some given amount of time on the user's machine, then the iteration count will automatically increase with any hardware speed improvement. This will negate all advantage that a brute-force attacker might gain from computers becoming faster, if we make the natural assumption that technological progress benefits password verifiers at the same rate as password crackers.

4. (Resource maximization) User-side parallelism is exploited to raise the cost of attacks.

Users care about (real) elapsed time; attackers about cumulative CPU time. Independently of the idea of hiding the iteration count, we design the key derivation to be parallelizable even for a single key. With the popularization of multi-core PCs, users will then be able to increase the total cost of key derivation without increasing the observed elapsed time that matters to them. The heightened total cost is however borne in full by the adversary, who gains nothing by parallelizing "within" single passwords, as opposed to "across" several ones.

These benefits are complementary rather than independent: for example, by accentuating the iteration unpredictability, Properties 2 and 3 solidify the Property 1 security gains that ride on it. Property 4 is orthogonal, but is equally crucial to our goal of making attacks maximally expensive.

**User Acceptance.** Aside from the technical arguments we develop in the remaining of this paper, remains the question of user acceptance. Although we cannot answer this question in the name of others, it seems reasonable to assume that acceptance should be easy.

The general principle of using deliberately expensive cryptography in conjunction with passwords has become standard, and is expected by users. The main commercial operating systems even use login screens that frustrate casual password guessing "by hand" using fake delays. Although this theatre provides but illusory protection against true offline attacks, it eloquently demonstrates that users (or system provisioners) *demand* that penalties be assessed for entering bad passwords. HKDFs fulfil these expectations in a cryptographically sound way, but in stark contrast to those commercial approaches, HKDFs seek to empower users without burdening them, for their benefit.

## 1.5 Related Work

The first deliberate use of expensive cryptographic operations to slow down brute-force attacks, in the `crypt()` password hashing function on Unix systems, coincides with the public availability of the DES cipher. Since then, a lot of progress has been made.

Provos and Mazières [33] have proposed a cost-parameterizable alternative to Unix `crypt()`, called `bcrypt()`, to avoid the obsolescence problems associated with fixed iteration counts. In their proposal, the cost parameter is set by the system administrator, shared among users, and must be committed to storage (rather than kept secret, set arbitrarily, and easy to program using the user interface, in the present work). More recently, Halderman *et al.* [16] proposed the idea of making key derivation very slow the first time, and subsequently faster by caching some state on the user machine: this is mostly useful for client-server authentication when the password is so weak that online trial-and-error is the greater concern, seconded by cache exposure. Interestingly, online PAKE protocols [27] have recently started to take offline dictionary attacks into consideration, by avoiding keeping user passwords in the clear on the server, and by distributing these servers among several locations. Other approaches to password management seek to prevent dictionary attacks in specific contexts: the PwdHash [36] system is a browser plug-in

that generates reproducible unique passwords for different web sites, and offers some resistance to both online and offline attacks.

Deliberately expensive cryptography has also been applied in “proof-of-work” schemes for combatting junk email [14] as well as for carrying out micro-payments [2], among other similar applications. These CPU-bound constructions are based on easy-to-verify but hard(er)-to-compute answers to random challenges built from hash functions. Memory-bound proof-of-work schemes have also been proposed [13], motivated not by the desire to prevent parallelism, but rather by the observation that memory chips have narrower and more predictable speed ranges than CPUs. At the other extreme of this spectrum, time-lock puzzles [35] are encryption schemes designed to be decryptable, without a key, after a well-defined but very long computation; these schemes are based on algebraic techniques, and view the publicity of the decryption delay as a feature [28].

Regarding parallel hashing schemes, we mention Split MAC [41], which is a parallelizable version of HMAC [24] for hashing long messages (rather than a long loop from a short password). On the cryptanalytic side, we mention Hellman’s [18] classic time/space trade-off attack against deterministic password hashing, and its modern reincarnation as Oeschlin’s [30] rainbow tables. See also [3] for a theoretical study of these types of algorithms.

**Contribution.** The point of this paper is as much to study HKDFs for their own sake as a new cryptographic and security tool, as it is to advocate their deployment in all practical systems that do password-based encryption.

In Section 2 we define HKDFs, construct them generically, and prove their basic security. We also parameterize them for the long term, and discuss user-side parallelism. In Section 3 we adopt a theoretical stance and study the origin of the  $\sim 4\times$  security factor that seems to arise magically.

In Section 4 we put on a systems hat and show how to integrate HKDFs in popular software such as *TrueCrypt* and *GnuPG*. We plan to release our implementations as open-source C code.

## 2 HKDF Design

The guiding design principles of Halting Key Derivation Functions are the following:

1. the cost of key derivation is programmed by the user and has no prior upper bound;
2. the amount of work for each key is independent and secret;

3. the key derivation memory footprint grows in lock-step with computation time;
4. the computation for deriving a single key can be distributed if needed.

We have already mentioned the motivation for (1.) letting the user program the iteration count  $t$  arbitrarily, and (4.) providing user-side parallelism. The justification for (2.) keeping  $t$  a secret, and (3.) having the memory footprint grow linearly with  $t$ , are to force the attacker to make costly guesses as it tries out wrong candidate passwords from its dictionary  $D$ .

Suppose the adversary is certain the true password  $w$  belongs in  $D$ , but has no idea about  $t$ . The obvious approach is to try out all the words in  $D$ , in parallel, for as many iterations as needed. However, this attack is incredibly memory-consuming since for each word there is state to be kept: terabytes or more for mere 40-bit entropy passwords ( $\#D = 2^{40}$ ).

If the attacker cannot maintain state across all of  $D$  as the iteration count is increased, the only alternative is to fix an upper bound  $\bar{t}$  for  $t$  and try each word for  $\bar{t}$  iterations, and then start over with a bigger  $\bar{t}$ . Clearly, this is more expensive since much of the computations is being redone. How much more expensive depends on the schedule for increasing  $\bar{t}$ . Increase it too slowly, *e.g.*,  $\bar{t} = 1, 2, 3, \dots$ , and most of the work ends up being redone. Increase it too fast, *e.g.*,  $\bar{t} = 1!, 2!, 3!, \dots$ , and the true value of  $t$  risks being overshoot by a wide margin.

We shall see that with the optimal strategy the attacker can keep the cost as low as  $\sim 4\times$  as much as if  $t$  has been public. The user does not pay this penalty since on the correct password the HKDF halts spontaneously at the correct iteration count  $t$  (which the user need not recall either). This gives us  $\sim 2$  bits of extra security essentially *for free*.

The memory footprint growth in  $\Theta(t)$  is a technicality to ensure that the argument holds for arbitrarily large  $t$ , lest it become more economical beyond some threshold to purchase the memory.

## 2.1 Formal Specification

As briefly outlined in Section 1.4, an HKDF consists of a pair of deterministic functions:

**Prepare** :  $(w, r, t) \mapsto v$  which, given a password  $w$ , a random string  $r$ , and an iteration count  $t$ , produces a public verification string  $v$ ;

**Extract** :  $(w, v) \mapsto k$  which, given a password  $w$  and a verification string  $v$ , outputs a key  $k$  upon halting, or fails to halt in polynomial time.

In this abstract model, the iteration count parameter  $t$  is given to **Prepare** at the onset. In practice, the user sets  $t$

implicitly by interrupting the computation as she pleases, using the user interface.

**Security Model.** We write  $[a]$  and  $[a \mid b]$  to denote marginal and conditional distributions of random variables. Let  $\mathcal{U}_S$  denote the uniform distribution over a set  $S$ , often implicit from context.

Pick  $r \in_s \{0, 1\}^\ell$ , viz., so that  $[r] \equiv \mathcal{U}_{\{0, 1\}^\ell}$ , to be our  $\ell$ -bit random seed for some parameter  $\ell$ . We first demand that the extracted keys be uniform and statistically independent of the secrets:

- Key uniformity:  $[k \mid w, t] \equiv \mathcal{U}$  where  $k = \text{Extract}(w, \text{Prepare}(w, r, t))$ .

We also impose lower and upper computational complexity bounds on the functions:

- Preparation complexity:  $\text{Prepare}(w, r, t)$  always halts in time  $O(t)$ , for all inputs.
- Extraction complexity:  $\text{Extract}(w, v)$  requires time and space  $\Theta(t)$ , for all  $v = \text{Prepare}(w, r, t)$ .
- Conditional halting:  $\text{Extract}(w', v)$  does not halt in polynomial time when  $w' \neq w$ .

We then ask that the key be unknowable without the requisite effort, even with all the data:

- Bounded indistinguishability:  $[v, k \mid w, t] \stackrel{o(t)}{\equiv} \mathcal{U}$  for  $v = \text{Prepare}(w, r, t)$  and  $k = \text{Extract}(w, v)$ .

I.e., for any randomized algorithm running in space (and hence time) strictly sub-linear in  $t$ , the joint  $[v, k]$  is computationally indistinguishable from random even given  $w$  and/or  $t$ .

As a consequence of the latter, the public string  $v$  is computationally indistinguishable from random to anyone who has not also guessed (and tested for  $t$  iterations) the correct password against it.

To summarize, for random  $r$ , it must be infeasible to find, in polynomial time in the security parameter, a tuple  $(k, t, w, w')$  such that  $k = \text{Extract}(w', \text{Prepare}(w, r, t))$  and  $w \neq w'$ . Furthermore, finding a tuple  $(k, t, w)$  such that  $k = \text{Extract}(w, \text{Prepare}(w, r, t))$  must require  $\Theta(t)$  units of time and memory, barring which no information about the correct  $k$  must be obtained from  $w, r, t$ .

## 2.2 Generic Construction

There are many ways to realize HKDFs, depending on the computational assumptions we make. One of the simplest constructions is generic and is based on some cryptographic hash function  $H : \{0, 1\}^{2\ell} \rightarrow \{0, 1\}^\ell$  viewed as a random oracle, for a security parameter  $\ell$ .

To capture the main idea, we start with a sequential HKDF construction. The construction is:

### HKDF<sup>H</sup>.Prepare( $w, r, t$ )

Inputs: password  $w$ , random string  $r$ , iteration count  $t$  (may be implicit from user interrupt).

Output: verification string  $v$  (and corresponding key  $k$ ).

1.  $z \leftarrow H(w, r)$  // init  $z$  from password and seed
2. FOR  $i := 1, \dots, t$  or until interrupted //
3.  $y_i \leftarrow z$  // store  $z$  in array element  $y_i$
4. REPEAT  $q$  times //
5.  $j \leftarrow 1 + (z \bmod i)$  // map  $z$  to some  $j \in \{1, \dots, i\}$
6.  $z \leftarrow H(z, y_j)$  // update  $z$
7.  $v \leftarrow (H(y_1, z), r)$  //
8.  $k \leftarrow H(z, r)$  //

### HKDF<sup>H</sup>.Extract( $w, v$ )

Inputs: password  $w$ , verification string  $v$ .

Output: derived key  $k$ , or may never halt.

0. parse  $v$  as  $(h, r)$  // comparison and seed strings
1.  $z \leftarrow H(w, r)$  //
2. FOR  $i := 1, \dots, \infty$  // forever loop
3.  $y_i \leftarrow z$  //
4. REPEAT  $q$  times //
5.  $j \leftarrow 1 + (z \bmod i)$  //
6.  $z \leftarrow H(z, y_j)$  //
7. IF  $H(y_1, z) = h$  THEN BREAK // break on halting condition
8.  $k \leftarrow H(z, r)$  //

The constant  $q$  is a parameter that determines the ratio between the time and space requirements. Since the Extract function may not halt spontaneously, it must be resettable by the user interface.

## 2.3 Security Properties

It is easy to see that the key output by Prepare is random and correctly reproducible by Extract. As for the HKDF security properties, we state the following lemmas.

**Lemma 1.** Key uniformity:  $[k \mid w, t] \equiv \mathcal{U}_{\{0, 1\}^\ell}$  where  $k = \text{Extract}(w, \text{Prepare}(w, r, t))$ .

**Lemma 2.** Preparation complexity:  $\text{Prepare}(w, r, t)$  halts in time  $\Theta(qt)$  on all inputs, for fixed  $q$ .

**Lemma 3.** Extraction complexity:  $\text{Extract}(w, v)$  halts in time  $\Theta(qt)$  and uses  $\Theta(t)$  bits of memory, for any  $v = \text{Prepare}(w, r, t)$  with same  $w$ .

*Proofs.* Since  $r$  is random and  $H$  is a random function,  $k = H(z, r)$  is uniformly distributed  $\forall z$ , which establishes Lemma 1. Lemmas 2 and 3 follow by inspection of the algorithms.  $\square$

**Lemma 4.** *Conditional halting:* Except with negligible probability,  $\text{Extract}(w', v)$  halts in super-polynomial time  $\Omega(2^\ell q)$  for any  $v = \text{Prepare}(w, r, t)$  and  $w' \neq w$ , where the probability is taken over the random choice of  $H$  for arbitrary inputs.

*Proof.* For  $w' \neq w$ , the value of  $y_1 = H(w, r)$  in  $\text{Prepare}$  and  $y'_1 = H(w', r)$  in  $\text{Extract}$  will be statistically independent since  $H$  is a random function, and therefore so will be the benchmark  $h = H(y_1, z)$  and its comparison value  $H(y'_1, z')$  for all  $z'$ . Since the constant  $h$ , the variable  $z'$ , and the value  $H(y'_1, z')$ , are all  $\ell$ -bit binary strings, we find that, letting  $\ell \rightarrow \infty$ ,

$$\Pr(\text{Extract loops indefinitely}) = e^{-1} \approx 0.3678794,$$

$$\Pr(\text{Extract halts before count } i) = (1 - e^{-1})(1 - e^{-i/2^\ell}).$$

The probability of halting on the wrong password in sub-exponential time  $i < 2^{o(\ell)}$  is negligible.  $\square$

**Lemma 5.** *Bounded indistinguishability:* the distributions  $[v, k \mid w, t]$  and  $\mathcal{U}_{\{0,1\}^{3\ell}}$  are perfectly indistinguishable by any algorithm running in sub-linear time and/or space  $o(t)$  in the iteration count, for any  $v = \text{Prepare}(w, r, t)$  and  $k = \text{Extract}(w, v)$ .

*Informal proof sketch.* We deal with the time-bound indistinguishability claim first. Observe that both  $v$  and  $k$  are independent outputs of a chain of  $qt$  applications of  $H$ , seeded by  $r$ . Since  $H$  is a random oracle, a standard argument shows that no information about  $(v, k)$  can be obtained without  $qt$  queries to  $H$ , which establishes the time-bound indistinguishability claim.

For the stronger space-bound indistinguishability claim, a more subtle argument shows that, with overwhelming probability, all possible computation paths require that “almost all”  $y_i$  for  $i = 1, \dots, t$  be stored in memory. The argument is based on the following sequence of observations: (1) For all  $i \in \{1, \dots, t\}$  and all  $i' \in \{i, \dots, t\}$ , the value  $y_i$  computed at step  $i$  will be needed at a subsequent step  $i'$  with probability  $\Pr(y_i \text{ needed at step } i') = q/i'$ , independently of its prior uses. (2) The expected number of times that  $y_i$  will be needed in the course of the entire computation is  $\#\{i' : y_i \text{ needed at step } i'\} \doteq \sum_{i'=i+1}^t (q/i') \approx q \ln(t/i)$ , which is  $\geq nq$  for any  $n > 0$  and  $i \leq e^{-n}t$ . (3) The probability that for fixed  $i \leq e^{-n}t$  the value  $y_i$  is never needed is  $\Pr(y_i \text{ not needed}) \leq e^{-nq}$ , which whenever  $n > \ell/(q \ln 2)$  is a vanishingly small function of the effective security parameter  $\ell$ . (4) Since, for

such  $n$ , the difference  $e^{-n}t - e^{-n-1}t$  is a linear function of  $t$ , the sub-linear memory constraint requires that some  $y_j$  with  $j \leq e^{-n-1}t$  be dropped prior to reaching the  $\lceil e^{-n}t \rceil$ -th step. (5) With overwhelming probability  $\Pr \geq 1 - e^{-nq}$ , the dropped value  $y_j$  appears in the computation path of some  $y_i$  where  $j < e^{-n}t < i$ , and without the value of  $y_j$  the key derivation cannot proceed.

The outcome of this reasoning is that before we can compute  $y_i$ , we need to recompute the dropped value  $y_j$ , which itself requires the recomputation of some earlier values still: some of these values must also have been dropped, as the same reasoning shows using an incremented  $n \leftarrow n + 1$  (with recursion upper bound  $\lfloor \ln t \rfloor$ ). To complete the argument, we note that for some  $l$  where  $j < l < i \leq t$ , the recomputation of  $y_j$  needed for  $y_i$  will require freeing up some previously stored value  $y_l$ , which is still needed for the calculation of  $y_i$ , and whose recomputation will require  $y_j$ ; when this happens, the algorithm will be stuck. This shows that the *intrinsic* space complexity of computing  $\text{HKDF}^H.\text{Extract}$  by whatever means in the random oracle model is  $\Theta(\ell t)$ .  $\square$

A consequence of Lemma 5 is that, unless the attacker has an enormous and linearly increasing amount of memory at its disposal, it will not be able to mount a “*persistent*” attack against all  $D$  (or any significant fraction thereof). It will have to choose which bits of state must be kept, and which ones must be erased to make room for others: the attack will necessarily be “*forgetful*”.

## 2.4 Parallelizable Construction

In addition to allowing arbitrarily large  $t$  and forcing the adversary to guess it, a complementary way to increase the adversary’s workload is to exploit any parallelism that is available to the user. Indeed, users care about the *real elapsed time* for processing a *single password*, whereas attackers worry about the *total CPU time* needed to cycle through the *entire dictionary*. Hence, we can hurt the adversary by increasing the CPU-time/elapsed-time ratio, with parallelizable key derivation.

Interestingly, we note that this runs contrary to conventional wisdom on password hashing, which traditionally abhors parallelism. The reason why our new *password-level parallelism* is safe is that only the legitimate user can benefit from it. The adversary is always better off using the cruder kind of *dictionary-level parallelism* that has always been available to him.

We require a cryptographic hash function  $H : \{0, 1\}^* \rightarrow \{0, 1\}^\ell$  where  $\ell$  is a security parameter. Let  $\{\text{STATEMENT}(l)\}_{l=1, \dots, p}$  denote the  $p$  independent statements  $\text{STATEMENT}(1), \dots, \text{STATEMENT}(p)$ , where  $p$  is a

“maximum parallelism” parameter. Our generic parallelizable HKDF is as follows:

$p\text{HKDF}^H.\text{Prepare}(w, r, t)$

Inputs: password  $w$ , random string  $r$ , iteration count  $t$  (may be implicit from user interrupt).

Output: verification string  $v$  (and corresponding key  $k$ ).

```

1.  $\{z_l \leftarrow H(w, r, l)\}_{l=1, \dots, p}$  // init each  $z_l$ 
                                     // independently
2.  $z \leftarrow H(z_1, \dots, z_l)$  // init  $z$  from all
                                     // the  $z_l$ 
3. FOR  $i := 1, \dots, t$  or until interrupted //
4.    $y_i \leftarrow z$  // store  $z$  in array
                                     // element  $y_i$ 
5. REPEAT  $q$  times //
6.    $\{j_l \leftarrow 1 + (z_l \bmod i)\}_{l=1, \dots, p}$  // map each  $z_l$  to
                                     // some  $j_l \in \{1, \dots, i\}$ 
7.    $\{z_l \leftarrow H(z_l, y_{j_l}, l)\}_{l=1, \dots, p}$  // update each  $z_l$ 
                                     // independently
8.    $z \leftarrow H(z_1, \dots, z_l)$  // update  $z$ 
9.    $v \leftarrow (H(y_1, z), r)$  //
10.   $k \leftarrow H(z, r)$  //
```

$p\text{HKDF}^H.\text{Extract}(w, v)$

Inputs: password  $w$ , verification string  $v$ .

Output: derived key  $k$ , or may never halt.

```

0. parse  $v$  as  $(h, r)$  //
1.  $\{z_l \leftarrow H(w, r, l)\}_{l=1, \dots, p}$  //  $p$ -way
                                     // parallelizable
2.  $z \leftarrow H(z_1, \dots, z_l)$  //
3. FOR  $i := 1, \dots, \infty$  //
4.    $y_i \leftarrow z$  //
5. REPEAT  $q$  times //  $p$ -way
                                     // parallelizable
                                     // across whole
                                     // loop
6.    $\{j_l \leftarrow 1 + (z_l \bmod i)\}_{l=1, \dots, p}$  //  $p$ -way
                                     // parallelizable
7.    $\{z_l \leftarrow H(z_l, y_{j_l}, l)\}_{l=1, \dots, p}$  //  $p$ -way
                                     // parallelizable
8.    $z \leftarrow H_0(z_1, \dots, z_l)$  //
9.   IF  $H(y_1, z) = h$  THEN BREAK //
10.   $k \leftarrow H(z, r)$  //
```

The constant  $p$  determines the maximum parallelizability of the scheme: it can then vary from 1-fold to  $p$ -fold without significant overhead. Total computational cost is  $\Theta(pqt)$  hash evaluations. Total memory requirement is  $\Theta(p + t)$  hash values, including a constant  $\ell p$  bits of memory overhead compared to the basic construction. Complexity-wise, the parameter  $p$  acts as a multiplier on the space/time proportionality ratio  $q$ , so that all security properties are retained with  $pq$  instead of  $q$ . It is thus

easy to enable parallelism by increasing  $p$  and decreasing  $q$  proportionately.

The relative penalty exerted on the adversary will be proportional to the number  $N$  of CPUs that the user can bring to bear, under the constraint that  $N \leq p$  (and where ideally,  $N \mid p$ ).

**Partitioned Memory.** The sequential scheme of Section 2.2 can also be made  $p$ -wise parallelizable for  $p = 2^l$ , by dropping  $l$  bits from  $r$  when **Prepare**-ing the public string  $v = (h, r)$ . To re-derive the key, the user tries all completions of  $r$  by running  $p$  instances of **Extract** at once until one halts. With  $p$  machines, the elapsed time is unchanged; however the total work is  $\Theta(pqt)$ . The inconvenient is that this requires  $\Theta(pt)$  memory instead of  $\Theta(p + t)$  for the method of Section 2.4, but the advantage is that processing and memory can be partitioned over  $p$  independent machines. Applying the same trick to the Section 2.4 scheme, gives us a hybrid with two parallelism options.

## 2.5 Practical Parameters

HKDF parameter selection is non-critical and much easier than with regular KDFs, since we are not trying to make decisions for the user, or prevent obsolescence by betting *pro* or *con* Moore’s law. The only choices we need to make concern the coefficients  $p$  and  $q$ . The rule of thumb is: maximize  $pq$  in view of today’s machines, and then fix  $p$  to cover all foreseeable needs for parallelism.

For the sake of illustration, let  $\ell = 256$ , and suppose that the user’s key derivation hardware can compute  $n = 2^{25}$  hashes per second (e.g., with  $2^3$  cores each capable of  $2^{22}$  hashes per second), and suppose the device has  $m = 2^{21} \cdot 256$  bits = 64 MiB of shared memory. Memory capacity will be reached after  $T = mpq/\ell n$  seconds of elapsed computation time. Thus, if we aim for  $pq = 2^{20}$ , the maximum selectable processing time on the device will be  $2^{16}$  seconds (close to 1 day), in increments of  $2^{-5}$  second. We can take  $p = 2^{10} \cdot 3^2 \cdot 5^2 = 230\,400$  and hence  $q = 4$  to get  $pq \approx 2^{20}$ . Last, we ascertain that, per all these choices, the available memory is still much larger than the  $\ell p \approx 7$  MiB of overhead that are the price to pay for the parallelization option.

Suppose then that the user settles for  $t = 2^5$  iterations (to take 1 second on the current device), and chooses a weak password with only 40 bits of entropy (from an implicit dictionary of size  $d = 2^{40}$ ). In these conditions, an adversary will need  $\ell t d = 2^{53}$  bits = 1024 TiB of memory in order to conduct a persistent attack. On a faster and/or more highly parallelized device, the user would choose a correspondingly larger value of  $t$ , further increasing the load on the adversary.

**Flexible Parallelism.** It is advisable to set  $p$  as a large product of small factors, to facilitate the even distribution of workload among any number  $N$  of CPUs such that  $N$  divides  $p$ ; this is easy to achieve in practice since the values of  $pq$  tend to be quite large, on the order of  $pq \geq 1\,000\,000$ . A nice consequence is that the same HKDF can be dimensioned to accommodate any reasonably foreseen amount of user-side parallelism (hence the choice  $p = 2^{10} \cdot 3^2 \cdot 5^2 = 230\,400$ ), and still be usable on today's sequential computers (with at least  $\ell p \approx 7$  MiB of memory in this example).

### 3 The Security Gap

We show that any adversary lacking enormous amounts of memory will incur a  $\sim 4\times$  larger cost for not knowing the iteration count. Since the penalty only strikes on wrong guesses, the user who knows the correct password will be immune to it. We say that *HKDFs widen the “security gap”*.

#### 3.1 Offline Dictionary Attack Model

We consider the simplest and most general offline attack by an adversary  $\mathcal{A}$  against a challenger  $\mathcal{C}$ . We capture the password “guessability” by supposing that it is drawn uniformly at random from a known dictionary  $D$ , and define its entropy as the value  $\log_2(\#D)$ . The game is as follows:

**Challenge.** The challenger  $\mathcal{C}$  picks  $w \in_{\mathcal{S}} D$  and  $r \in_{\mathcal{S}} \{0,1\}^{\ell}$  at random, chooses  $t \in \mathbb{N}$ , and computes  $(v, k) \leftarrow \text{HKDF.Prepare}(w, r, t)$ . It gives the string  $v$  to  $\mathcal{A}$ .

**Attack.** The adversary  $\mathcal{A}$  outputs as many keys as it pleases, sequentially:  $k_1, k_2, \dots$ . It wins the game as soon as some  $k_i$  matches  $k = \text{HKDF.Extract}(w, v)$ .

We assume that  $\mathcal{A}$  can only retain state for a dwindling fraction of  $D$ , of size  $o(1)$  in  $t$ .

**Password (Min-)Entropy.** In reality, passwords are not sampled uniformly from a fixed  $D$ , but rather non-uniformly from a set with no clear boundaries. The worst-case unpredictability of a password chosen in this manner is the *minimum entropy*, or *min-entropy*, defined as  $-\log_2(\max_w \Pr(w))$ . The uniform password model  $w \in_{\mathcal{S}} D$  conveniently and accurately reflects the difficulty of guessing from  $\mathcal{C}$ 's true password distribution, provided that  $\log_2(\#D)$  matches the min-entropy of the latter.

#### 3.2 Finding the Optimal Attack Strategy

By Lemma 5 we know that  $\mathcal{A}$  cannot do better than outputting random keys until it “tries out” the correct password  $w$  for  $t$  iterations (using the **Extract** function). Since  $\mathcal{A}$  lacks the memory to maintain concurrent instances of **Extract** for any substantial subset of  $D$ , the only option is to “dovetail” the search, *i.e.*:

- try all the words of  $D$  one by one (or few by few) for a bounded stretch of time;
- retry the same for longer and longer time stretches, until  $t$  is eventually exceeded.

We can neglect the  $o(1)$  fraction of  $D$  on which  $\mathcal{A}$  could run a persistent attack. Also, for uniform  $w \in_{\mathcal{S}} D$  and unknown  $t$  it is easy to show that it is optimal to spend the same amount of effort on each candidate password. We deduce that the optimal algorithm for any *forgetful* attacker  $\mathcal{A}$  is:

Optimal-MemoryBound- $\mathcal{A}_D(v)$

Input: verification string  $v$ .

Output: password  $\hat{w}$  and key  $\hat{k}$ .

1. FOR  $\hat{t} := t_1, t_2, \dots$  //  $t_1 < t_2 < \dots$   
: the search  
schedule
2. FOR  $\hat{w} \in D$  // in sequence  
or partially  
parallel
3. RUN  $\hat{k} \leftarrow \text{Extract}(\hat{w}, v)$  for  $\hat{t}$  steps
4. IF  $\hat{k} \in \{0, 1\}^{\ell}$  THEN // did Extract  
halt sponta-  
neously?
5. RETURN  $(\hat{w}, \hat{k})$  //

The only parameters to be specified are the increasing sequence of iteration counts  $t_1 < t_2 < \dots$ ; the optimal schedule  $(t_1, t_2, \dots)$  will depend on  $\mathcal{A}$ 's uncertainty on  $t$ .

**Effort and Penalty.** We now quantify the total computation effort expended by  $\mathcal{A}$  in function of the attack schedule  $(t_1, t_2, \dots)$ . Let us denote by  $\mathcal{W}_{t_1, t_2, \dots}^{\mathcal{A}}(t)$  the total expected number of hash evaluations made by  $\mathcal{A}$  is the iteration count chosen by  $\mathcal{C}$  is  $t$ . Let  $k$  be the smallest index such that  $t_k \geq t$ . Let  $d = \#D$ , and define the constant  $u = dq$ . Since all of  $D$  will be explored for each  $t_i < t$ , and only half of  $D$  on expectation for the first  $t_k \geq t$  (and nothing thereafter), we find that:

$$\mathcal{W}_{t_1, t_2, \dots}^{\mathcal{A}}(t) = \left( \sum_{i=1}^{k-1} t_i + \frac{t_k}{2} \right) u,$$

where  $\begin{cases} k = \min\{i : t_i \geq t\} \\ u = (\#D)q \end{cases}$ .

If, on the other hand,  $\mathcal{A}$  had known the value of  $t$  and just had to search for the password alone, the expected attack effort, denoted  $\mathcal{W}_t^{\mathcal{A}}(t)$ , would have been:

$$\mathcal{W}_t^{\mathcal{A}}(t) = \left(\frac{t}{2}\right) u, \quad \text{with } u = (\#D)q.$$

We define the *penalty* (of not knowing  $t$ ) as the ratio:  $\pi(t) = \frac{\mathcal{W}_{t_1, t_2, \dots}^{\mathcal{A}}(t)}{\mathcal{W}_t^{\mathcal{A}}(t)} = \frac{2(t_1 + \dots + t_{k-1}) + t_k}{t} \geq 1$ . Next, we show how to bound  $\pi(t)$ .

### 3.3 Bounding the Uncertainty Penalty

First, we should clarify that the goal of  $\mathcal{A}$  is to minimize the value of  $\pi(t)$  *on expectation* over the random choices made by  $\mathcal{A}$  and  $\mathcal{C}$ , and not necessarily in the extremal cases where  $t$  is either very small or very large. Indeed, it is not in the interest of  $\mathcal{C}$  to choose too small a value for  $t$ . Furthermore,  $\mathcal{A}$  can easily achieve  $\pi(t) = 1$  for the maximal value of  $t$  (we assume that  $\mathcal{A}$  knows what hardware  $\mathcal{C}$  uses), simply by setting  $t_1 = t_{\max}$ , but this would be a Pyrrhic victory since the attack would be utterly prohibitive and probably for naught. More generally,  $\mathcal{A}$  cannot simply let  $t_1$  be the largest “likely” value for  $t$ , since then  $\mathcal{C}$  would figure it out and select  $t = t_1 + 1$ .

The foregoing strongly suggests that the game-theoretic optimum must be *scale-invariant* over the entire range  $\{t_{lo}, \dots, t_{hi}\} \ni t$  that  $\mathcal{C}$  considers useful. It also suggests that  $\mathcal{A}$  and  $\mathcal{C}$  should use mixed (*i.e.*, randomized) strategies. We use the notation  $[V]$  to denote the distribution of  $V$ .

**Lemma 6. Uniform equilibrium:** There exists a constant  $\pi_0$ , function of  $t_{lo}$  and  $t_{hi}$ , such that a Nash equilibrium between  $\mathcal{A}$  and  $\mathcal{C}$  can only be reached for a randomized attack strategy such that  $\forall t \in \{t_{lo}, \dots, t_{hi}\} : \pi(t) = \pi_0$ . The corresponding optimal strategy for  $\mathcal{A}$  exists.

*Informal proof sketch.* Let  $[(t_1, t_2, \dots)]$  be an optimal mixture, or distribution of schedules, for  $\mathcal{A}$ , and suppose toward a contradiction that for this strategy the expected penalty  $\pi(t)$  is not uniform over the entire range of acceptable values for  $t$ . Thus, there exist  $t_{\text{easy}}$  and  $t_{\text{hard}}$  in the interval  $\{t_{lo}, \dots, t_{hi}\}$  such that  $\forall t : \pi(t_{\text{easy}}) \leq \pi(t) \leq \pi(t_{\text{hard}})$ . Since the mixture is optimal,  $\mathcal{C}$  can compute its parameters and select  $t = t_{\text{hard}}$  to exert the stiffest expected penalty on  $\mathcal{A}$ . Predicting this,  $\mathcal{A}$  would let  $\rho = t_{\text{hard}}/t_{\text{easy}}$  and switch to a new mixture given by:  $[(t'_1, t'_2, \dots)] = [(\rho t_1, \rho t_2, \dots)]$ .

It is easy to see that  $\pi'(t) = \pi'(t_{\text{hard}})$  under the new mixture equals  $\pi(t_{\text{easy}}) < \pi(t)$  under the original one. It follows that the new strategy performs better than the old one when  $\mathcal{C}$  consistently chooses  $t = t_{\text{hard}}$  (which was  $\mathcal{C}$ 's optimal defense in response to  $\mathcal{A}$ 's supposedly optimal

attack). It follows that the original strategy was not optimal after all, and we conclude that any optimal randomized attack must incur the same penalty  $\pi_0 = \pi(t)$  for all  $t \in \{t_{lo}, \dots, t_{hi}\}$ , as claimed. Existence of the randomized strategy characterized above follow from Nash.  $\square$

**Lemma 7. Scale invariance:** In the limit  $(t_{hi}/t_{lo}) \rightarrow \infty$ , the optimal attack and defense strategies are scale-invariant. For  $\mathcal{A}$  the optimal ratio  $[(t_{i+1}/t_i)]$  converges in distribution to a mixture  $[\alpha]$  that is independent of  $i$ . For  $\mathcal{C}$  the optimal parameter  $[t]$  assumes a Zipf power law, whose probability density function  $\frac{d}{dx}\text{Pr}(t < x)$  is proportional to  $x^{-\beta}$  for some negative exponent  $-\beta$  in the limit.

*Informal proof sketch.* Consider an optimal mixed strategy  $[(t_1, t_2, \dots)]$  and an iteration count  $t$ . Without loss of generality, we assume that  $t_{lo} \ll t \ll t_{hi}$ . Fix some  $\delta > 0$ , and let  $t' = (1 + \delta)t$ . By Lemma 6, we know that  $\pi(t) = \pi(t')$ . Now, consider the mixed schedule  $[(t'_1, t'_2, \dots)]$  obtained by substituting  $t'_i = (1 + \delta)t_i$  for  $t_i$  everywhere, while keeping all probabilities the same. Denote by  $\pi'$  the penalty function under that new schedule. By definition, we have the identity  $\pi'(t') = \frac{1+\delta}{1+\delta} \pi(t) = \pi(t)$ , and by transitivity we obtain that  $\pi(t) = \pi'(t)$ . We conclude that  $\pi(t) = \pi'(t)$  for any distribution of  $t$  over the interval  $\{(1 + \delta)t_{lo}, \dots, (1 + \delta)^{-1}t_{hi}\}$ , for any  $\delta > 0$ .

Since the strategy is optimal, it follows that multiplying all the values in all the schedules it comprises by any constant  $(1 + \delta)$  must preserve  $\pi(t)$ ; this also works backward for  $(1 + \delta)^{-1}$ , and thus this is true in the limit for any multiplier in  $\mathbb{R}^+$ . In other words an optimal strategy for  $\mathcal{A}$  is invariant to (multiplicative) scaling. A straightforward argument then shows that this must be reciprocated by the optimal response employed by  $\mathcal{C}$ . Approximating  $t$  as a real in  $\mathbb{R}^+$ , we deduce that  $t$  must obey a Zipf power law, whose density is:  $\frac{d}{dx}\text{Pr}(t < x) \propto x^{-\beta}$  for some  $\beta \in \mathbb{R}$ .

For the remaining claim, we first note that the scale invariance implies that all the individual schedules  $(t_1, t_2, \dots)$  in the mixture must satisfy  $(t_{i+1}/t_i) = (t_{j+1}/t_j)$  for all  $i, j$ , otherwise the multiplication by a constant would result in a different mixture. We have not yet ruled out the possibility of (sub-)mixtures  $[(t_1, t_2, \dots)], [(t'_1, t'_2, \dots)], \dots$  with unequal progressions  $[(t_{i+1}/t_i)] \neq [(t'_{i+1}/t'_i)]$ , which is why so far we say that  $[(t_{i+1}/t_i)]$  converges to a distribution  $[\alpha]$  instead of a value  $\alpha^*$ .  $\square$

**Randomized Starting Point.** Lemmas 6 and 7 show that the optimal attack schedule for  $\mathcal{A}$  is a randomized sequence  $(t_1, t_2, \dots)$  where  $t_i = t_1 \alpha^{i-1}$  for some random starting point  $t_1 \approx t_{lo}$  and a progression coefficient

$\alpha \in_s [\alpha]$ . For large enough  $t \gg t_1$ , the penalty becomes:

$$\pi(t) = \left( 2 \frac{t_1 + \dots + t_{k-1}}{t} + \frac{t_k}{t} \right) \approx \frac{\alpha + 1}{\alpha - 1} \gamma, \\ \text{for some } \gamma = \frac{t_k}{t} \in [1, \alpha].$$

Applying the scale invariance principle, we know that the expected  $\pi(t)$  should be constant for varying  $t$ , which requires that  $\gamma$  be distributed with density  $\propto \gamma^{-1}$ :

$$\frac{d}{dx} \Pr(\gamma < x) = \begin{cases} x^{-1}/\ln \alpha & \text{for } 1 \leq x < \alpha \\ 0 & \text{otherwise} \end{cases}. \quad (1)$$

Since  $\gamma$  has expectation  $\int_1^\alpha x \frac{x^{-1}}{\ln \alpha} dx = \frac{\alpha-1}{\ln \alpha}$ , the uniform penalty for all choices of  $t$  is thus:

$$\pi(t) \approx \pi_0 = \frac{\alpha + 1}{\ln(\alpha)}. \quad (2)$$

**Optimal Progression Coefficient.** The last thing we need is to compute  $\pi_0$  in function of the progression coefficient  $\alpha$ , which is drawn from some distribution  $[\alpha]$  yet to be specified. Notice from Equation (2) that  $\pi_0$  is a convex function of  $\alpha$  that reaches a minimum for some  $\alpha = \alpha^* \in (1, \infty)$ , hence the optimal  $[\alpha]$  is the pointwise distribution centered on  $\alpha^*$ . Asymptotically, the numerical values of the optimal attack coefficient  $\alpha^*$  and the corresponding minimal penalty  $\pi_0^*$  are given by:

$$\alpha^* = \arg \min_{\alpha} \frac{\alpha + 1}{\ln(\alpha)}, \quad \pi_0^* = \frac{\alpha^* + 1}{\ln(\alpha^*)}, \\ \pi_0^* = \alpha^* \approx 3.59112147666862. \quad (3)$$

To implement the optimal strategy, a rational attacker  $\mathcal{A}$  would fix  $\alpha = \alpha^*$  from Equation (3), and start the search schedule from some random  $t_1 = t_{i_0} \gamma$  where  $\gamma$  is distributed as in Equation (1). No matter how cleverly  $\mathcal{C}$  chooses  $t$ , the expected penalty incurred by  $\mathcal{A}$  is  $\pi(t) = \pi_0^* \approx 3.5911215$ . (We mention that the same constant, 3.59112..., arises in the context of the cow-path problem [23], which is a hidden search problem with a related structure and also with scale-invariance properties.)

Finally, and reciprocally, we can determine the optimal Zipf-Pareto exponent  $-\beta^*$  that a rational  $\mathcal{C}$  should choose to oppose  $\mathcal{A}$ . Straightforward calculations show that  $-\beta^* = -1$ .

### 3.4 Justifying the Zipf-Pareto Hypothesis

We have shown that (for the stated objective of maximizing the expected security gap) the optimal distribution  $[t]$  is a power law of exponent  $-\beta^* = -1$  over some fixed and fairly wide interval of interest. The question is whether this is a reasonable assumption to make for the behavior of  $\mathcal{C}$ :

*Would a typical user not always program the same key derivation delay?*

The first answer to this question depends on the user's psychology, and his or her understanding of the benefits provided by HKDFs. In fact, it is sufficient that the attacker *believe* that the user has a good reason to use very long delays on occasion (*e.g.*, to protect a particularly sensitive ciphertext, or to shield a long-term backup password that will only be used as a last resort, as we already discussed in the introduction). Of course, if the attacker does not believe such a thing, but the user does it anyway, it is the attacker who will be sorry.

The second answer is a phenomenological one. Zipf or Pareto distributions (of law  $\propto n^{-\beta}$ ) have been noted to occur ubiquitously in the upper tail of empirical distributions in a variety of contexts, ranging from physics and geology with the distribution of reserves in oil field deposits, to linguistics with the relative frequency of words in written texts, to economics regarding distribution of income and normalized returns of securities, and even to anthropology with the size of human population centers (see [34] for a list of these phenomena). Hence, it seems natural to assume that for large ensembles of users and/or ciphertexts, the induced iteration count  $t$  would be akin to a Zipf process. This hypothesis draws credence from an observed pattern in natural and human sciences [31, 25] that the most common empirical distributions are Zipf-Pareto of exponent  $-\beta^* = -(1 + \epsilon) \lesssim -1$ .

In summary, we have shown that the HKDF approach gives us a small amount of “free” security:

**Theorem 8. Security gain:** Under the reasonable hypothesis that users do not always choose  $t$  predictably, HKDFs increase the “effective entropy” of any password, over regular KDFs, by:

$$\log_2(\pi_0^*) \approx 1.84443445579378 \text{ bits}.$$

## 4 Real-world Implementation

We believe that the case is strong for dropping KDFs in favor of HKDFs wherever possible, and to make it even stronger we discuss two compelling real-world applications.

We present two implementations of HKDFs on GNU/Linux systems, which we intend to release as open-source portable (POSIX) C code. Our first prototype is as a stand-alone command-line tool to be used in conjunction with programs such as GNU gpg [15] or Ruusu's aespipe [1] to assemble strong password-based encryption pipelines. Our second prototype is a patch for the truecrypt [40] “plausibly deniable” disk encryption software, which dramatically increases its resistance to offline dictionary attacks, and thus plausible deniability by implication.

We will see that HKDFs are much more secure in practice than the KDFs they replace, at the cost of little tweaks to the UI, minimal impact on the user behavior, and no change to the hardware.

## 4.1 TrueCrypt Disk Encryption

*TrueCrypt* [40] is a password-based disk encryption software of modern design, developed for *Windows* and subsequently ported to *Linux*, and available under a permissive open-source license. *TrueCrypt* is aimed at local storage encryption underneath the filesystem. It provides *plausible deniability*, meaning that a *truecrypt*-encrypted disk should be indistinguishable from a *shred*-ded disk to anyone who lacks the password. Free-space ciphertext *and* plaintext are designed to look random: this allows a nested volume to be hidden in a container volume's free space. This is perhaps the central feature of the design, and *truecrypt* is able to avoid clobbering the hidden volume when writing on the container, as long as both volumes are mounted.

The cryptographic design is otherwise fairly standard. A password-based KDF-encrypted header holds a randomly generated key, needed for encrypting the bulk of the data (*i.e.*, the disk sectors). One peculiarity is that the KDF iteration count cannot be recorded because the encrypted volume *including the header* must appear random, and so it is burned into the *truecrypt* binary.

**Plausible Deniability.** Had they been available, HKDFs would have been very helpful, indeed:

1. There would be no need to record the iteration count anywhere, yet no reason to keep it fixed.
2. Plausible deniability would be *enhanced greatly*, because it all hinges on the password and the effort needed to crack it, and we know that HKDFs make that much harder for at least three reasons (arbitrarily large counts, widened security gap, and user-side parallelism).

**Implementation and Interface.** *TrueCrypt*'s KDF is PKCS#5 with a user-selected hash (SHA1, RIPEMD160, or WHIRLPOOL) and a hard-coded iteration count (2000, 2000, and 1000 respectively). Since the hash selection cannot be recorded in the volume any more than the iteration count, *truecrypt* simply tries the three functions in sequence until one works.

We implemented the generic HKDF of Section 2.2, instantiated with SHA1, as a fourth option to be tried last (quite naturally, since it must be allowed to run for an arbitrary amount of time). The encrypted volume format has space for 64 bytes of PKCS#5 salt; we reclaim 40

bytes for the HKDF public string *v* (which *is* random, see Section 2), and pad the rest with random data.

The two functions we interface with *TrueCrypt* are:

```
ulong HKDF_prepare(    //returns: actual value of t
    ulong tmax,        //input: maximum t, 0 for none
    uint  w_sz, uchar const *w, //input: password w
    uint  r_sz, uchar const *r, //input: randomness r
    uint  v_sz, uchar *v,      //output: public string v
    uint  k_sz, uchar *k);     //output: derived key k

ulong HKDF_derive(     //returns: actual value of t
    ulong tmax,        //input: maximum t, 0 for none
    uint  w_sz, uchar const *w, //input: password w
    uint  v_sz, uchar const *v, //input: public string v
    uint  k_sz, uchar *k);     //output: derived key k
```

We build a modified version of *TrueCrypt*, called *hkdf-tc*, that invokes *HKDF\_prepare()* when asked to create a new volume with the HKDF option turned on, and defers to *HKDF\_derive()* when asked to mount a volume with an undecipherable header. Although both functions take a parameter *tmax* that could play the role of *t*, the actual selection of *t* is implicit and interactive:

*When creating a new volume*, *hkdf-tc* asks the user to enter the same password twice, and to choose a number of options. If the HKDF option is selected, *HKDF\_prepare()* will invite the user to press a key after any—short or long—delay, explaining that the same delay will be incurred every time the volume is mounted as a defense against password guessers.

*When mounting an existing volume*, *hkdf-tc* queries the password and tries the built-in KDFs. If these fail, *HKDF\_derive()* is invoked, and the user instructed to press a key if it is taking too long, for the program cannot distinguish a wrong password from one with a longer delay.

In both cases, computations proceed in the background, pending the user signal which is detected by polling a non-blocking I/O system call at every iteration of the main loop. At  $\sim 1\text{--}30$  Hz, this solution is responsive but not wasteful, and fits well with *TrueCrypt*'s command-line user interface.

With a graphical UI, another approach would be to add a button to the password entry dialog, greyed out at first, and becoming clickable once the user has entered a password: its label when commissioned by *HKDF\_prepare()* would be [finish]; or [cancel] when commissioned by *HKDF\_derive()*. One could also add a busy indicator, progress bar, or iteration counter, to taste.

## 4.2 Command-line HKDF Tool

Our second implementation is a small command-line tool, called *hkdf*, whose usage is as follows:

1. `hkdf -p [-r|-s] [-t MAX]` prompts for a passphrase, and prints a public string
2. `hkdf -k [-r|-s] [-t MAX] FILE` same, but writes `publ.str.` to `FILE` and prints the key
3. `hkdf [-d] [-t MAX]` reads `publ.str.`, asks for a passphr., and prints a key

Arguments: `-p|-k` PREPARE mode --- once running, press the \* key to finish  
`-d` EXTRACT mode (the default) --- press Control-C to cancel  
`-r` reads randomness from `stdin` (instead of `/dev/random`)  
`-s` reads passphrase from `stdin` (instead of user prompt)  
`-t MAX` triggers auto-finish or auto-cancel at iteration `MAX`

Each of the following commands creates a random public string `v`, saves it to the file `public.v`, and prints the corresponding key `k` on standard output, based on the user's passphrase. The second command asks for the passphrase twice (on behalf of `hkdf -p` and `hkdf -d`, in unspecified order), and re-derives `k` on-the-fly to provide end-to-end verification without committing any secret to disk.

```
# hkdf -k public.v
# hkdf -p | tee public.v | hkdf -d
```

The user must press the \* key at some time after entering the passphrases(s) (or use the `-t` option) to set the key derivation delay. To recover `k` from `public.v` at a later time, we use:

```
# hkdf < public.v
```

which prompts for the passphrase once.

**Encryption with AESpipe and GnuPG.** We can combine `hkdf` with `aespipe` [1] to assemble a (randomized) password-based AES encryptor with HKDF resistance to dictionary attacks. The plaintext is a file `plain.bin` and the ciphertext will consist of two files `crypt.v` and `crypt.aes`. To encrypt:

```
# aespipe -p 4 4<<<'hkdf -k crypt.v' \
    < plain.bin > crypt.aes
```

In the Bourne shell (`/bin/sh`), the string `4<<<'...'` causes the command between the backquotes to be executed in a sub-shell, and its output redirected to the parent's unused file descriptor #4; meanwhile, the parameter `-p 4` instructs `aespipe` to fetch its key from the same. To decrypt:

```
# aespipe -d -p 4 4<<<'hkdf < crypt.v' \
    < crypt.aes > decrypted.bin
```

This command works similarly. If the passphrase is good, `hkdf` will feed the right key to `aespipe`; otherwise, it will run forever until interrupted by Control-C.

The `hkdf` tool is even easier to interface with other programs, e.g., `gpg` [15]:

```
# hkdf -k crypt.v | gpg --passphrase-fd 0 \
    -o crypt.gpg -c plain.bin
# hkdf < crypt.v | gpg --passphrase-fd 0 \
    -o decrypted.bin crypt.gpg
```

This is merely suggestive; more sophisticated scripts could merge the ciphertext into a single file.

**GnuPG Key-rings.** Since the user passphrase is the Achilles' heel of the system, an excellent use of the `hkdf/gpg` synergy is to replace `gpg`'s default keyring encryption with something stronger. To quote the `gpg(1)` manual page:

#### WARNINGS

*Use a \*good\* password for your user account and a \*good\* passphrase to protect your secret key. This passphrase is the weakest part of the whole system. Programs to do dictionary attacks on your secret keyring are very easy to write and so you should protect your "/.gnupg/" directory very well.*

HKDFs are an excellent way to add protection with or without changing the passphrase. Our `hkdf` tool and a small script to bind it to `gpg` are all that is needed.

### 4.3 Concrete Security Gains

We now quantify the security gained by upgrading *TrueCrypt* and *GnuPG* from KDF to HKDF. Our test platform is a 1.5 GHz single-core x86 laptop running *Debian Linux*.

**Baseline Measurements.** First we clock the various built-in KDFs to establish the benchmark: cf. Table 1. These timings were obtained by instrumenting the relevant sections of code, in order to suppress overheads and obtain an accurate indication of the amount of work needed for a brute-force attack.

**HKDF Performance.** Next, we measure the performance of the HKDF implementation, and the rate at which the size of the state is increased: cf. Table 2. As we would expect, the raw throughput is very close to but slightly less than a "pure" implementation of the corresponding hash function (e.g., compare the SHA1 instantiation with `gpg` above). The discrepancy is caused by the modular reduction in the inner loop of the HKDF algorithm.

**Attainable Security Gains.** We now find the actual key derivation complexity (time and space) for several user-programmed delays, and what this entails for an optimal attacker. We fix  $q = 57600$ : cf. Table 3. The last column shows the actual security gain provided by HKDFs in comparison to the benchmarks. For the most casual uses (where the HKDF preparation is finished without deliberate delay), we expect a steady security gain of about  $\sim 7$  bits over *TrueCrypt*, and about  $\sim 11$  bits over *GnuPG*. For more sensitive uses, gains of  $\sim 15$ – $20$  bits can be attained with a few minutes of patience. For long-term backups where two-hour waits can be justified, the gain reaches  $\sim 23$  bits over *GnuPG*. The security gain further increases by  $\sim \log_2(N)$  bits in all cases if the user’s machine has  $N$  CPUs.

To give a very concrete example, a *GnuPG* secret key file will be equally well protected with an 11-letter all-lowercase password ( $\sim 51$  bits of entropy) by *gpg* itself, as by our *hkdf* system with a 6-letter password ( $\sim 28$  bits of entropy) plus a two-hour wait—or eight-minute on a sixteen-core machine. An infrastructure the scale of Google ( $\sim 10^5$  CPUs) would take two years to crack either.

**Attack Times.** Our last table compares the times to crack one password in *GnuPG*, *TrueCrypt*, and various HKDF use cases, in function of the password strength (40 and 60 bits of entropy, or  $\sim 9$  and  $\sim 13$  random lowercase letters, respectively), against a spectrum of opponents: cf. Table 4.

Even with “instantaneous” user delays ( $\sim 1$  s), the security gains are substantial and may suffice to turn a successful attack into a successful defense. Larger delays ( $> 1$  min.–1 hr.) are surprisingly secure with the inherent benefits of HKDFs; they are justified for last-resort disaster-recovery backups, which must remain secure, and their passwords not forgotten, over long cryptoperiods.

## References

- [1] AESpipe - AES encrypting or decrypting pipe. <http://loop-aes.sourceforge.net/>.
- [2] BACK, A. Hashcash. Technical report, 1997. <http://www.cypherspace.org/hashcash/>.
- [3] BARKAN, E., BIHAM, E., AND SHAMIR, A. Rigorous bounds on cryptanalytic time/memory trade-offs. In *Advances in Cryptology—CRYPTO 2006*.
- [4] BELLARE, M., POINTCHEVAL, D., AND RO-GAWAY, P. Authenticated key exchange se-cure against dictionary attacks. In *Advances in Cryptology—EUROCRYPT 2000*.
- [5] BELLOVIN, S. M., AND MERRITT, M. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE Symposium on Security and Privacy—SP 1992*.
- [6] BROWN, D. R. L. Prompted user retrieval of se-cret entropy: The passmaze protocol. Cryptology ePrint Archive, Report 2005/434, 2005. <http://eprint.iacr.org/>.
- [7] CALLAS, J., DONNERHACHE, L., FINNEY, H., AND THAYER, R. OpenPGP message format. RFC 2440, November 1998. <http://www.ietf.org/rfc/rfc2440.txt>.
- [8] CANETTI, R., HALEVI, S., KATZ, J., LINDELL, Y., AND MACKENZIE, P. Universally compos-able password-based key exchange. In *Advances in Cryptology—EUROCRYPT 2005*.
- [9] CANETTI, R., HALEVI, S., AND STEINER, M. Mitigating dictionary attacks on password-protected local storage. In *Advances in Cryptology—CRYPTO 2006*. Full version: Cryptology ePrint Archive, Report 2006/276. <http://eprint.iacr.org/>.
- [10] CryptoCard. <http://www.cryptocard.com/>.
- [11] DEAN, D., AND STUBBLEFIELD, A. Using client puzzles to protect TLS. In *USENIX Security Symposium—SECURITY 2001*. <http://www.usenix.org/events/sec01/full-papers/dean/dean.pdf>.
- [12] DigiPass. <http://www.vasco.com/>.
- [13] DWORK, C., GOLDBERG, A., AND NAOR, M. On memory-bound functions for fighting spam. In *Advances in Cryptology—CRYPTO 2003*.
- [14] DWORK, C., AND NAOR, M. Pricing via pro-cessing or combating junk mail. In *Advances in Cryptology—CRYPTO 1992*.
- [15] GnuPG - the GNU privacy guard. <http://www.gnupg.org/>.
- [16] HALDERMAN, J. A., WATERS, B., AND FELTEN, E. W. A convenient method for securely managing passwords. In *Proceedings of WWW 2005*.
- [17] HALEVI, S., AND KRAWCZYK, H. Public-key cryptography and password protocols. In *ACM CCS 1998*.

- [18] HELLMAN, M. E. A cryptanalytic time-memory trade-off. *IEEE Trans. Information Theory* 26, 4 (1980), 401–6.
- [19] IEEE P1363.2: Password-based public-key cryptography. <http://grouper.ieee.org/groups/1363/>.
- [20] JABLON, D. Strong password-only authenticated key exchange. *Computer Communication Review* (1996).
- [21] JUELS, A., AND BRAINARD, J. Client puzzles: A cryptographic defense against connection depletion attacks. In *Proceedings of NDSS 1999*.
- [22] KALISKI, B. PKCS #5: Password-based cryptography specification, version 2.0. RFC 2898, September 2000. <http://www.ietf.org/rfc/rfc2898.txt>.
- [23] KAO, M.-Y., REIF, J. H., AND TATE, S. R. Searching in an unknown environment: An optimal randomized algorithm for the cow-path problem. In *ACM-SIAM Symposium on Discrete Algorithms—SODA 1993*.
- [24] KRAWCZYK, H., BELLARE, M., AND CANETTI, R. HMAC: Keyed-hashing for message authentication. RFC 2104, February 1997. <http://www.ietf.org/rfc/rfc2104.txt>.
- [25] LAHERRERE, J., AND SORNETTE, D. Stretched exponential distributions in nature and economy: 'fat tails' with characteristic scales. *European Physical Journals B2* (1998), 525–39. <http://xxx.lanl.gov/abs/cond-mat/9801293>.
- [26] LENSTRA, A. K., AND VERHEUL, E. R. Selecting cryptographic key sizes. *Journal of Cryptology* 14, 4 (2001), 255–93.
- [27] MACKENZIE, P., SHRIMPTON, T., AND JAKOBSSON, M. Threshold password-authenticated key exchange. *Journal of Cryptology* 19, 1 (2006), 27–66.
- [28] MAO, W. Send message into a definite future. In *Proceedings of ICICS 1999*.
- [29] NAOR, M., AND PINKAS, B. Visual authentication and identification. In *Advances in Cryptology—CRYPTO 1997*.
- [30] OECHSLIN, P. Making a faster cryptanalytical time-memory trade-off. In *Advances in Cryptology—CRYPTO 2003*.
- [31] PERLINE, R. Zipf's law, the central limit theorem, and the random division of the unit interval. *Physical Review E* 54, 1 (1996), 220–3.
- [32] PINKAS, B., AND SANDER, T. Securing passwords against dictionary attacks. In *ACM Conference on Computer and Communications Security—CCS 2002*.
- [33] PROVOS, N., AND MAZIÈRES, D. A future-adaptable password scheme. In *USENIX Technical Conference—FREENIX Track 1999*. <http://www.usenix.org/events/usenix99/provos/provos.pdf>.
- [34] REED, W. J. The Pareto, Zipf and other power laws. *Economics Letters* 74, 1 (2001), 15–9.
- [35] RIVEST, R. L., SHAMIR, A., AND WAGNER, D. A. Time-lock puzzles and timed-release crypto. Technical report MIT-LCS-TR-684, MIT, 1985. <http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-684.pdf>.
- [36] ROSS, B., JACKSON, C., MIYAKE, N., BONEH, D., AND MITCHELL, J. C. Stronger password authentication using browser extensions. In *USENIX Security Symposium—SECURITY 2005*.
- [37] RSA-LABORATORIES. PKCS #5: Password-based encryption standard, version 1.5, November 1993. See also [22].
- [38] SecurID. <http://www.rsasecurity.com/>.
- [39] STUBBLEFIELD, A., AND SIMON, D. Inkblot authentication. Tech. report MSR-TR-2004-85, Microsoft Research, 1985.
- [40] TrueCrypt - free open-source on-the-fly disk encryption software. <http://www.truecrypt.org/>.
- [41] VIEGA, J., KOHNO, T., AND HOUSLEY, R. Patent-free, parallelizable MACing. Crypto Forum Research Group, December 2002. <http://www1.ietf.org/mail-archive/web/cfrg/current/msg00126.html>.
- [42] VON AHN, L., BLUM, M., HOPPER, N., AND LANGFORD, J. CAPTCHA: Using hard AI problems for security. In *Advances in Cryptology—CRYPTO 2003*.
- [43] YAN, J., BLACKWELL, A., ANDERSON, R., AND GRANT, A. The memorability and security of passwords - some empirical results. *IEEE Security and Privacy* 2, 5 (2004), 25–31.

Table 1: Baseline measurements.

Software	Digest function	Normalized speed	Fixed multiplier	Time per password (as measured)
truecrypt	HMAC-SHA1	25200 #/s	2000 #	79 ms
	HMAC-RIPEMD160	20400 #/s	2000 #	98 ms
	HMAC-WHIRLPOOL	9700 #/s	1000 #	101 ms
gpg	MD5	30.0 MB/s	65536 B	2.2 ms
	SHA1 (default)	28.0 MB/s	65536 B	2.3 ms
	SHA256	15.2 MB/s	65536 B	4.3 ms
	SHA512	9.9 MB/s	65536 B	6.6 ms

Table 2: HKDF performance.

$H$ algorithm for HKDF <sup>H</sup>	Hash width $\ell$	HKDF throughput	Time resolution and Memory rate (@1 CPU)			
			$(q = 57600)$		$(q = 230400)$	
SHA1	160	25.1 MB/s	11.0 Hz	220 B/s	2.8 Hz	56 B/s
WHIRLPOOL	512	19.7 MB/s	2.7 Hz	173 B/s	0.7 Hz	45 B/s

Table 3: Attainable security gains.

Program	$H$ for HKDF <sup>H</sup>	Time & Memory (per password) :				Security gain vs. built-in KDF
		Programmed		Adversarial		
hkdf-tc (vs. truecrypt)	WHIRLPOOL	3 sec.	< 1 kB	11 sec.	2 kB	$10^2 \times$ ( $\sim 7$ bits)
		4 min.	41 kB	14 min.	147 kB	$10^4 \times$ ( $\sim 13$ bits)
		45 min.	0.5 MB	3 hours	1.7 MB	$10^5 \times$ ( $\sim 17$ bits)
hkdf/gpg (vs. gpg)	SHA1	1 sec.	< 1 kB	4 sec.	1 kB	$10^3 \times$ ( $\sim 10$ bits)
		10 min.	131 kB	36 min.	469 kB	$10^6 \times$ ( $\sim 20$ bits)
		2 hours	1.6 MB	7 hours	5.5 MB	$10^7 \times$ ( $\sim 23$ bits)

Table 4: Attack times.

Opponent	# CPUs	GnuPG	TrueCrypt	HKDF			
				1-core		32-core	
40-bit secret				1 s	10 m	1 s	1 h
Individual	$10^1$	7.7 y	275 y	12.5 k y	7.5 M y	401 k y	1.4 G y
Corporation	$10^4$	67 h	101 d	13 y	7.5 k y	401 y	1.4 M y
Huge botnet	$10^7$	242 s	2.4 h	(31 h) <sup>†</sup>	7.5 y	(41 d) <sup>†</sup>	1.4 k y
“The World”	$10^{10}$	242 ms	8.6 s	(110 s) <sup>†</sup>	(18 h) <sup>†</sup>	(59 m) <sup>†</sup>	(147 d) <sup>†</sup>
<sup>†</sup> The flagged figures relate to a <i>persistent</i> attack, feasible for these parameters if the opponent has 1 GiB per CPU.							
60-bit secret							
Government	$10^6$	80 y	2.9 k y	131 k y	79 M y	4.2 M y	15 G y
“The World”	$10^{10}$	70 h	105 d	13 y	7.9 k y	420 y	1.5 M y

# Spamscatter: Characterizing Internet Scam Hosting Infrastructure

David S. Anderson   Chris Fleizach   Stefan Savage   Geoffrey M. Voelker  
*Collaborative Center for Internet Epidemiology and Defenses*  
*Department of Computer Science and Engineering*  
*University of California, San Diego*

## Abstract

Unsolicited bulk e-mail, or SPAM, is a means to an end. For virtually all such messages, the intent is to attract the recipient into entering a commercial transaction — typically via a linked Web site. While the prodigious infrastructure used to pump out billions of such solicitations is essential, the engine driving this process is ultimately the “point-of-sale” — the various money-making “scams” that extract value from Internet users. In the hopes of better understanding the business pressures exerted on spammers, this paper focuses squarely on the Internet infrastructure used to host and support such scams. We describe an opportunistic measurement technique called *spamscatter* that mines emails in real-time, follows the embedded link structure, and automatically clusters the destination Web sites using *image shingling* to capture graphical similarity between rendered sites. We have implemented this approach on a large real-time spam feed (over 1M messages per week) and have identified and analyzed over 2,000 distinct scams on 7,000 distinct servers.

## 1 Introduction

Few Internet security issues have attained the universal public recognition or contempt of unsolicited bulk email — SPAM. In 2006, industry estimates suggest that such messages comprise over 80% over all Internet email with a total volume up to 85 billion per day [15, 17]. The scale of these numbers underscores the prodigious delivery infrastructures developed by “spammers” and in turn motivates the more than \$1B spent annually on anti-spam technology. However, the engine that drives this arms race is not spam itself — which is simply a means to an end — but the various money-making “scams” (legal or illegal) that extract value from Internet users.

In this paper, we focus on the Internet infrastructure used to host and support such scams. In particular, we

analyze spam-advertised Web servers that offer merchandise and services (e.g., pharmaceuticals, luxury watches, mortgages) or use malicious means to defraud users (e.g., phishing, spyware, trojans). Unlike mail-relays or bots, scam infrastructure is directly implicated in the spam profit cycle and thus considerably rarer and more valuable. For example, a given spam campaign may use thousands of mail relay agents to deliver its millions of messages, but only use a single server to handle requests from recipients who respond. Consequently, the availability of scam infrastructure is critical to spam profitability — a single takedown of a scam server or a spammer redirect can curtail the earning potential of an entire spam campaign.

The goal of this paper is to characterize scam infrastructure and use this data to better understand the dynamics and business pressures exerted on spammers. To identify scam infrastructure, we employ an opportunistic technique called *spamscatter*. The underlying principle is that each scam is, by necessity, identified in the link structure of associated spams. To this end, we have built a system that mines email, identifies URLs in real time and follows such links to their eventual destination server (including any redirection mechanisms put in place). We further identify individual scams by clustering scam servers whose rendered Web pages are graphically similar using a technique called *image shingling*. Finally, we actively probe the scam servers on an ongoing basis to characterize dynamic behaviors like availability and lifetime. Using the *spamscatter* technique on a large real-time spam feed (roughly 150,000 per day) we have identified over 2,000 distinct scams hosted across more than 7,000 distinct servers. Further, we characterize the availability of infrastructure implicated in these scams and the relationship with business-related factors such as scam “type”, location and blacklist inclusion.

The remainder of this paper is structured as follows. Section 2 reviews related measurement studies similar in topic or technique. In Section 3 we outline the struc-

ture and lifecycle of Internet scams, and describe in detail one of the more extensive scams from our trace as a concrete example. Section 4 describes our measurement methodology, including our probing system, image shingling algorithm, and spam feed. In Section 5, we analyze a wide range of characteristics of Internet scam infrastructure based upon the scams we identify in our spam feed. Finally, Section 6 summarizes our findings and concludes.

## 2 Related work

Spamscatter is an opportunistic network measurement technique [5], taking advantage of spurious traffic — in this case spam — to gain insight into “hidden” aspects of the Internet — in this case scam hosting infrastructure. As with other opportunistic measurement techniques, such as backscatter to measure Internet denial-of-service activity [20], network telescopes and Internet sinks [32] to measure Internet worm outbreaks [19, 21], and spam to measure spam relays [27], spamscatter provides a mechanism for studying global Internet behavior from a single or small number of vantage points.

We are certainly not the first to use spam for opportunistic measurement. Perhaps the work most closely related to ours is Ramachandran and Feamster’s recent study using spam to characterize the network behavior of the spam relays that sent it [27]. Using extensive spam feeds, they categorized the network and geographic location, lifetime, platform, and network evasion techniques of spam relay infrastructure. They also evaluated the effectiveness of using network-level properties of spam relays, such as IP blacklists and suspect BGP announcements, to filter spam. When appropriate in our analyses, we compare and contrast characteristics of spam relays and scam hosts; some scam hosts also serve as spam relays, for example. In general, however, due to the different requirements of the two underground services, they exhibit different characteristics; scam hosts, for example, have longer lifetimes and are more concentrated in the U.S.

The Webb Spam Corpus effort harvests URLs from spam to create a repository of *Web spam* pages, Web pages created to influence Web search engine results or deceive users [31]. Although both their effort and our own harvest URLs from spam, the two projects differ in their use of the harvested URLs. The Webb Spam Corpus downloads and stores HTML content to create an offline data set for training classifiers of Web spam pages. Spamscatter probes sites and downloads content over time, renders browser screenshots to identify URLs referencing the same scam, and analyzes various characteristics of the infrastructure hosting scams.

Both community and commercial services consume

URLs extracted from spam. Various community services mine spam to specifically identify and track phishing sites, either by examining spam from their own feeds or collecting spam email and URLs submitted by the community [1, 6, 22, 25]. Commercial Web security and filtering services, such as Websense and Brightcloud, track and analyze Web sites to categorize and filter content, and to identify phishing sites and sites hosting other potentially malicious content such as spyware and keyloggers. Sites advertised in spam provide an important data source for such services. While we use similar data in our work, our goal is infrastructure characterization rather than operational filtering.

Botnets can play a role in the scam host infrastructure, either by hosting the spam relays generating the spam we see or by hosting the scam servers. A number of recent efforts have developed techniques for measuring botnet structure, behavior, and prevalence. Cook et al. [9] tested the feasibility of using honeypots to capture bots, and proposed a combination of passive host and network monitoring to detect botnets. Bächer et al. [23] used honeypots to capture bots, infiltrate their command and control channel, and monitor botnet activity. Rajab et al. [26] combined a number of measurement techniques, including malware collection, IRC command and control tracking, and DNS cache probing. The last two approaches have provided substantial insight into botnet activity by tracking hundreds of botnets over periods of months. Ramachandran and Feamster [27] provided strong evidence that botnets are commonly used as platforms for spam relays; our results suggest botnets are not as common for scam hosting.

We developed an image shingling algorithm to determine the equivalence of screenshots of rendered Web pages. Previous efforts have developed techniques to determine the equivalence of transformed images as well. For instance, the SpoofGuard anti-phishing Web browser plugin compares images on Web pages with a database of corporate logos [7] to identify Web site spoofing. SpoofGuard compares images using robust image hashing, an approach employing signal processing techniques to create a compressed representation of an image [30]. Robust image hashing works well against a number of different image transformations, such as cropping, scaling, and filtering. However, unlike image shingling, image hashing is not intended to compare images where substantial regions have completely different content; refinements to image hashing improve robustness (e.g., [18, 28]), but do not fundamentally extend the original set of transforms.

## 3 The life and times of an Internet scam

In this section we outline the structure and life cycle of Internet scams, and describe in detail one of the

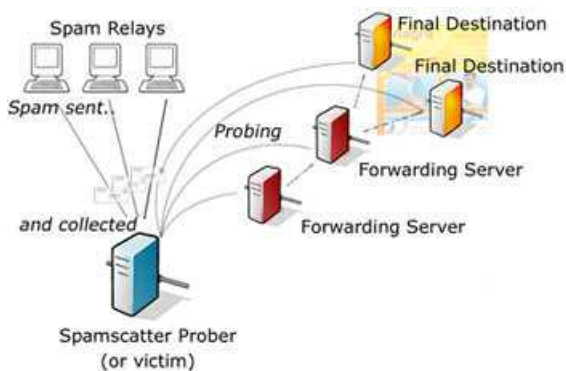


Figure 1: Components of a typical Internet scam.

more extensive scams from our trace as a concrete example. This particular scam advertises “Downloadable Software,” such as office productivity tools (Microsoft, Adobe, etc.) and popular games, although in general the scams we observed were diverse in what they offered (Section 5.1).

Figure 1 depicts the life of a spam-driven Internet scam. First, a spam campaign launches a vast number of unsolicited spam messages to email addresses around the world; a large spam campaign can exceed 1 billion emails [12]. In turn the content in these messages frequently advertises a *scam* — unsolicited merchandise and services available through the Web — by embedding URLs to scam Web servers in the spam; in our data, roughly 30% of spam contains such URLs (Section 5.1). An example of spam that does not contain links would be “pump-and-dump” stock spam intended to manipulate penny stock prices [3]; the recent growth of image-based stock spam has substantially reduced the fraction of spam using embedded URLs, shrinking from 85% in 2005 to 55% in 2006 [12]. These spam campaigns can be comparatively brief, with more than half lasting less than 12 hours in our data (Section 5.4). For our example software scam, over 5,000 spam emails were used to advertise it over a weeklong period.

Knowing or unsuspecting users click on URLs in spam to access content from the Web servers hosting the scams. While sometimes the embedded URL directly specifies the scam server, more commonly it indicates an intermediate Web server that subsequently redirects traffic (using HTTP or Javascript) on towards the scam server. Redirection serves multiple purposes. When spammer and scammer are distinct, it provides a simple means for tagging requests with the spammer’s affiliate identifier (used by third-party merchants to compensate independent “advertisers”) and laundering the spam-based origin before the request reaches the merchant (this laundering provides plausible deniability for the mer-



Figure 2: Screenshots, hostnames, and IP addresses of different hosts for the “Downloadable Software” scam. The highlighted regions show portions of the page that change on each access due to product rotation. Image shingling is resilient to such changes and identifies these screenshots as equivalent pages.

chant and protects the spammer from potential conflicts over the merchant’s advertising policy). If spammer and scammer are the same, a layer of redirection is still useful for avoiding URL-based blacklists and providing deployment flexibility for scam servers. In our traces, most scams use at least one level of redirection (Section 4).

On the back end, scams may use multiple servers to host scams, both in terms of multiple virtual hosts (e.g., different domain names served by the same Web server) and multiple physical hosts identified by IP address (Section 5.2). However, for the scams in our spam feed, the use of multiple virtual hosts is infrequent (16% of scams) and multiple physical hosts is rare (6%); our example software scam is one of the more extensive scams, using at least 99 virtual hosts on three physical hosts.

Finally, different Web servers (physical or virtual), and even different accesses to a scam using the same URL, can result in slightly different downloaded content for the same scam. Intentional randomness for evasion, rotating advertisements, featured product rotation, etc., add another form of aliasing. Figure 2 shows example screenshots among different hosts for the software scam. To overcome these aliasing issues, we use screenshots of Web pages as a basis for identifying all hosts participating in a given scam (Section 4.2).

A machine hosting one scam may be shared with other scams, as when scammers run multiple scams at once or the hosts are third-party infrastructure used by multiple scammers. Sharing is common, with 38% of scams be-

ing hosted on a machine with at least one other scam (Section 5.3). One of the machines hosting the software scam, for example, also hosted a pharmaceutical scam called “Toronto Pharmacy” (which happened to be hosted on a server in Guangzhou, China).

The lifetimes of scams are much longer than spam campaigns, with 50% of scams active for at least a week (Section 5.4). Furthermore, scam hosts have high availability during their lifetime (most above 99%) and appear to have good network connectivity (Section 5.5); the lifetime of our software scam ran for the entire measurement period and was available 97% of the time. Finally, scam hosts tend to be geographically concentrated in the United States; over 57% of scam hosts from our data mapped to the U.S. (Section 5.6.2). Such geographic concentration contrasts sharply with the location of spam relay hosts; for comparison, only 14% of spam relays used to send the spam to our feed are located in the U.S. Figure 3 shows the geographic locations of the spam relays and scam hosts for the software scam. The three scam hosts were located in China and Russia, whereas the 85 spam relays were located around the world in 30 countries.

The lifetimes, high availability, and good network connectivity, as well as the geographic diversity of spam relays compared with scam hosts, all reflect the fundamentally different requirements and circumstances between the two underground services. Spam relays require no interaction with users, need only be available to send mail, but must be great enough in number to mitigate the effects of per-host blacklists. Consequently, spam relays are well suited to “commodity” botnet infrastructure [27]; one recent industry estimate suggests that over 80% of spam is in fact relayed by bots [13]. By contrast, scam hosts are naturally more centralized (due to hosting a payment infrastructure), require interactive response time to their target customers, and may — in fact — be hosting legal commerce. Thus, scam hosts are much more likely to have high-quality hosting infrastructure that is stable over long periods.

## 4 Methodology

This section describes our measurement methodology. We first explain our data collection framework for probing scam hosts and spam relays, and then detail our image shingling algorithm for identifying equivalent scams. Finally, we describe the spam feed we use as our data source and discuss the inherent limitations of using a single viewpoint.

### 4.1 Data collection framework

We built a data collection tool, called the *spamscatter prober*, that takes as input a feed of spam emails, extracts the sender and URLs from the spam messages, and probes those hosts to collect various kinds of information (Figure 1). For spam senders, the prober performs a ping, traceroute, and DNS-based blacklist lookup (DNSBL) once upon receipt of each spam email. The prober performs more extensive operations for the scam hosts. As with spam senders, it first performs a ping, traceroute, and DNSBL lookup on scam hosts. In addition, it downloads and stores the full HTML source of the Web page specified by valid URLs extracted from the spam (we do not attempt to de-obfuscate URLs). It also renders an image of the downloaded page in a canonical browser configuration using the KHTML layout engine [14], and stores a screenshot of the browser window. For scam hosts, the prober repeats these operations periodically for a fixed length of time. For the trace in this paper, we probed each host and captured a screenshot while visiting each URL every three hours. Starting from when the first spam email introduces a new URL into the data set, we probe the scam host serving that URL for a week independently of whether the probes fail or succeed.

As we mentioned earlier, many spam URLs simply point to sites that forward the request onto another server. There are many possible reasons for the forwarding behavior, such as tracking users, redirecting users through third-party affiliates or tracking systems, or consolidating the many URLs used in spam (ostensibly to avoid spam filters) to just one. Occasionally, we also noticed forwarding that does not end, either indicating a misconfiguration, programming error, or a deliberate attempt to avoid spidering.

The prober accommodates a variety of link forwarding practices. While some links direct the client immediately to the appropriate Web server, others execute a series of forwarding requests, including HTTP 302 server redirects and JavaScript-based redirects. To follow these, the prober processes received page content to extract simple META refresh tags and JavaScript redirect statements. It then tracks every intermediate page between the initial link and the final content page, and marks whether a page is the end of the line for each link. Properly handling forwarding is necessary for accurate scam monitoring. Over 68% of scams used some kind of forwarding, with an average of 1.2 forwards per URL.

### 4.2 Image shingling

Many of our analyses compare content downloaded from scam servers to determine if the scams are equivalent. For example, scam hosts may serve multiple indepen-

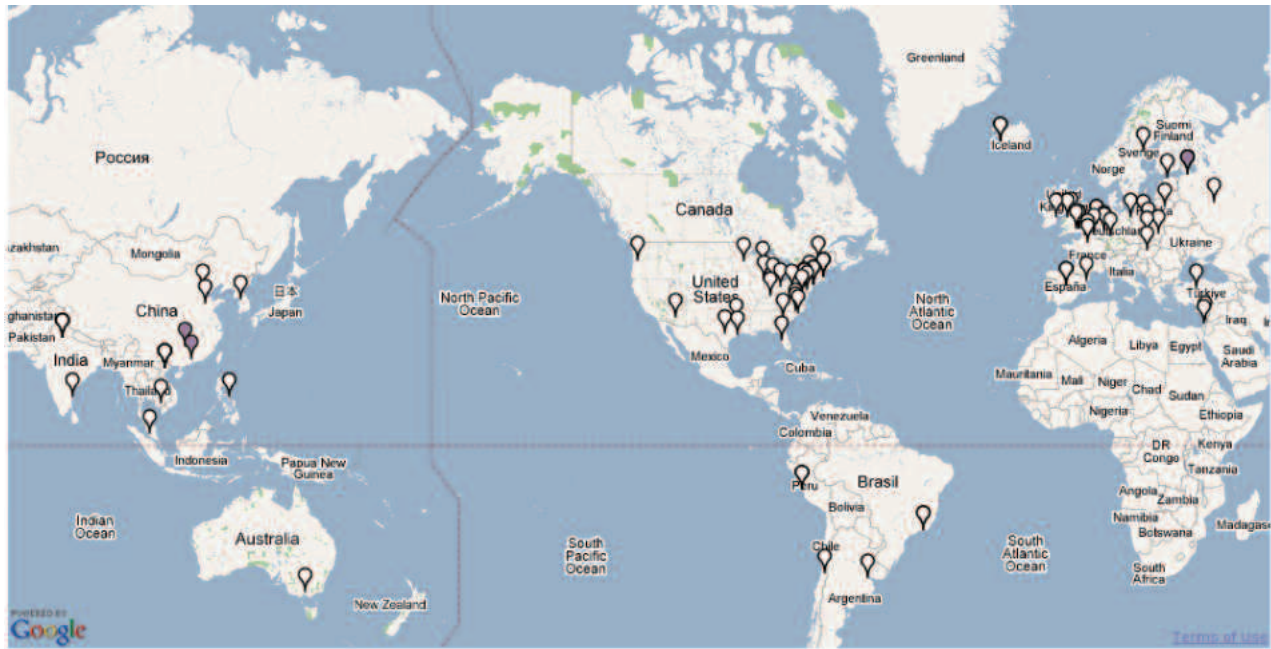


Figure 3: Geographic locations of the spam relays and scam server hosts for the “Downloadable Software” scam. The three scam servers are located in China and Russia and shown with dark grey points. The 85 spam relays are located around the world in more than 30 different countries, and are shown in white.

dent scams simultaneously, and we cannot assume that URLs that lead to the same host are part of the same scam. Similarly, scams are hosted on multiple virtual servers as well as distributed across multiple machines. As a result, we need to be able to compare content from scam servers on different hosts to determine whether they are part of the same scam. Finally, even for content downloaded from the same URL over time, we need to determine whether the content fundamentally changes (e.g., the server has stopped hosting the scam but returns valid HTTP responses to requests, or it has transitioned to hosting a different scam altogether).

Various kinds of aliasing make determining scam equivalence across multiple hosts, as well as over time, a challenging problem. One possibility is to compare spam messages within a window of time to identify emails advertising the same scam. However, the randomness and churn that spammers introduce to defeat spam filters makes it extremely difficult to use textual information in the spam message to identify spam messages for the same scam (e.g., spam filters continue to struggle with spam message equivalence). Another possibility is to compare the URLs themselves. Unfortunately, scammers have many incentives not to use the same URL across spams, and as a result each spam message for a scam might use a distinct URL for accessing a scam server. For instance, scammers may embed unique track-

ing identifiers in the query part of URLs, use URLs that contain domain names to different virtual servers, or simply randomize URLs to defeat URL blacklisting.

A third option is to compare the HTML content downloaded from the URLs in the spam for equivalence. The problem of comparing Web pages is a fundamental operation for any effort that identifies similar content across sites, and comparing textual Web content has been studied extensively already. For instance, text shingling techniques were developed to efficiently measure the similarity of Web pages, and to scale page comparison to the entire Web [4, 29]. In principle, a similar method could be used to compare the HTML text between scam sites, but in practice the downloaded HTML frequently provides insufficient textual information to reliably identify a scam. Indeed, many scams contained little textual content at all, and instead used images entirely to display content on the Web page. Also, many scams used frames, iframes, and JavaScript to display content, making it difficult to capture the full page context using a text-based Web crawler.

Finally, a fourth option is to render screenshots of the content downloaded from scam sites, and to compare the screenshots for equivalence. Screenshots are an attractive basis for comparison because they sidestep the aforementioned problems with comparing HTML source. However, comparing screenshots is not without

its own difficulties. Even for the same scam accessed by the same URL over time — much less across different scam servers — scam sites may intentionally introduce random perturbations of the page to prevent simple image comparison, display rotating advertisements in various parts of a page, or rotate images of featured products across accesses. Figure 2 presents an example of screenshots from different sites for the same scam that show variation between images due to product rotation.

Considering the options, we selected screenshots as the basis for determining spam equivalence. To overcome the problems described earlier, we developed an image-clustering algorithm, called image shingling, based on the notion of shingling from the text similarity literature. Text shingling decomposes a document into many segments, usually consisting of a small number of characters. Various techniques have been developed to increase the efficiency and reduce the space complexity of this process [11]. Next, these hashed “shingles” are sorted so that hashes for documents containing similar shingles are close together. The ordering allows all the documents that share an identical shingle to be found quickly. Finally, documents are clustered according to the percentage of shared shingles between them. The power of the algorithm is that it essentially performs  $O(N^2)$  comparisons in  $O(N \lg N)$  time.

Our image shingling algorithm applies a similar process to the image domain. The algorithm first divides each image into fixed size chunks in memory; in our experiments, we found that an image chunk size of 40x40 pixels was an effective tradeoff between granularity and shingling performance. We then hash each chunk to create an image shingle, and store the shingle on a global list together with a link to the image (we use the MD4 hash to create shingles due to its relative speed compared with other hashing algorithms). After sorting the list of shingles, we create a hash table, indexed by shingle, to track the number of times two images shared a similar shingle. Scanning through the table, we create clusters of images by finding image pairs that share at least a threshold of similar images.

To determine an appropriate threshold value, we took one day’s worth of screenshots and ran the image shingling algorithm for all values of thresholds in increments of 1%. Figure 4 shows the number of clusters created per threshold value. The plateau in the figure starting at 70% corresponds to a fair balance between being too strict, which would reduce the possibility of clustering nearly similar pages, and being too lenient, which would cluster distinct scams together. Manually inspecting the clusters generated at this threshold plateau and the cluster membership changes that occur at neighboring threshold values, we found that a threshold of 70% minimized false negatives and false positives for determining scam page

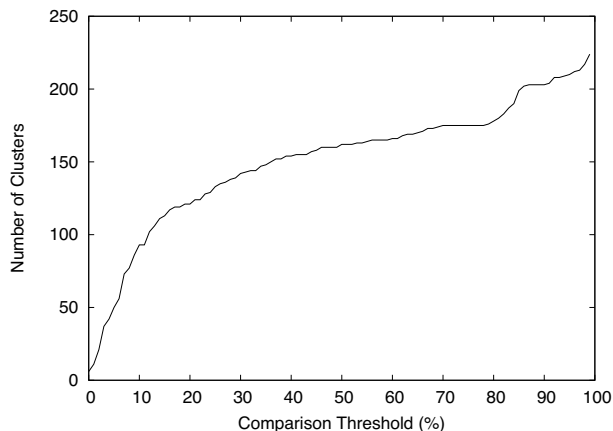


Figure 4: The choice of a threshold value for image shingling determines the number of clusters.

equivalence.

We have developed a highly optimized version of this basic algorithm that, in practice, completes an all-pairs comparison in roughly linear time. In practice, image shingling is highly effective at clustering similar scam pages, while neatly side-stepping the adversarial obfuscations in spam messages, URLs, and page contents. Clearly, a determined scammer could introduce steps to reduce the effectiveness of image shingling as described (e.g., by slightly changing the colors of the background or embedded images on each access, changing the compression ratio of embedded images, etc.). However, we have not witnessed this behavior in our trace. If scammers do take such steps, this methodology will likely need to be refined.

### 4.3 Spam feed and limitations

The source of spam determines the scams we can measure using this methodology. For this study, we have been able to take advantage of a substantial spam feed: all messages sent to any email address at a well-known four-letter top-level domain. This domain receives over 150,000 spam messages every day. We can assume that any email sent to addresses in this domain is spam because no active users use addresses on the mail server for the domain. Examining the “From” and “To” addresses of spam from this feed, we found that spammers generated “To” email addresses using a variety of methods, including harvested addresses found in text on Web pages, universal typical addresses at sites, as well as name-based dictionary address lists. Over 93% of “From” addresses were used only once, suggesting the use of random source addresses to defeat address-based spam blacklists.

<i>Characteristic</i>	<i>Summary Result</i>
Trace period	11/28/06 – 12/11/06
Spam messages	1,087,711
Spam w/ URLs	319,700 (30% of all spam)
Unique URLs	36,390 (11% of all URLs)
Unique IP addresses	7,029 (19% of unique URLs)
Unique scams	2,334 (6% of unique URLs)

Table 1: Summary of spamscluster trace.

We analyze Internet scam hosting infrastructure using spam from only a single, albeit highly active, spam feed. As with other techniques that use a single network viewpoint to study global Internet behavior, undoubtedly this single viewpoint introduces bias [2, 8]. For example, the domain that provides our spam feed has no actual users who read the email. Any email address harvesting process that evaluates the quality of email addresses, such as correlating spam email targets with accesses on scam sites, would be able to determine that sending spam to these addresses yields no returns (that is, until we began probing).

While measuring the true bias of our data is impossible, we can anecdotally gauge the coverage of scams from our spam feed by comparing them with scams identified from an entirely different spam source. As a comparison source, we used the spam posted to the Usenet group `news.admin.net-abuse.sightings`, a forum for administrators to contribute spam [22]. Over a single 3-day period, January 26–28th, 2007, we collected spam from both sources. We captured 6,977 spam emails from the newsgroup and 113,216 spam emails from our feed. The newsgroup relies on user contributions and is moderated, and hence is a reliable source of spam. However, it is also a much smaller source of spam than our feed.

Next we used image shingling to distill the spam from both sources into distinct scams, 205 from the newsgroup and 1,687 from our feed. Comparing the scams, we found 25 that were in both sets, i.e., 12% of the newsgroup scams were captured in our feed as well. Of the 30 most-prominent scams identified from both feeds (in terms of the number of virtual hosts and IP addresses), ten come from the newsgroup feed. These same ten, furthermore, were also in our feed. Our goal was not to achieve global coverage of all Internet scams, and, as expected, we have not. The key question is how representative our sample is; without knowing the full set of scams (a very challenging measurement task), we cannot gauge the representativeness of the scams we find. Characterizing a large sample, however, still provides substantial insight into the infrastructure used to host scams. And it is further encouraging that many of the most extensive scams in the newsgroup feed are also found in ours. Moving forward, we plan to incorporate other sources of

<i>Scam category</i>	<i>% of scams</i>
Uncategorized	29.57%
Information Technology	16.67%
Dynamic Content	11.52%
Business and Economy	6.23%
Shopping	4.30%
Financial Data and Services	3.61%
Illegal or Questionable	2.15%
Adult	1.80%
Message Boards and Clubs	1.80%
Web Hosting	1.63%

Table 2: Top ten scam categories.

spam to expand our feed and further improve representativeness.

## 5 Analysis

We analyze Internet scam infrastructure using scams identified from a large one-week trace of spam messages. We start by summarizing the characteristics of our trace and the scams we identify. We then evaluate to what extent scams use multiple hosts as distributed infrastructure; using multiple hosts can help scams be more resilient to defenses. Next we examine how hosts are shared across scams as an indication of infrastructure reuse. We then characterize the lifetime and availability of scams. Scammers have an incentive to use host infrastructure that provides longer lifetimes and higher availability; at the same time, network and system administrators may actively filter or take down scams, particularly malicious ones. Lastly, we examine the network and geographic locations of scams; again, scammers can benefit from using stable hosts that provide high availability and good network connectivity.

Furthermore, since spam relay hosts are an integral aspect of Internet scams, where appropriate in our analyses we compare and contrast characteristics of spam relays and scam hosts.

### 5.1 Summary results

We collected the spam from our feed for a one-week period from November 28, 2006 to December 4, 2006. For every URL extracted from spam messages, we probed the host specified by the URL for a full week (independent of whether the host responded or not) starting from the moment we received the spam. As a result, the prober monitored some hosts for a week beyond the receipt of the last spam email, up until December 11. Table 1 summarizes the resulting spamscluster trace. Starting with over 1 million spam messages, we extracted

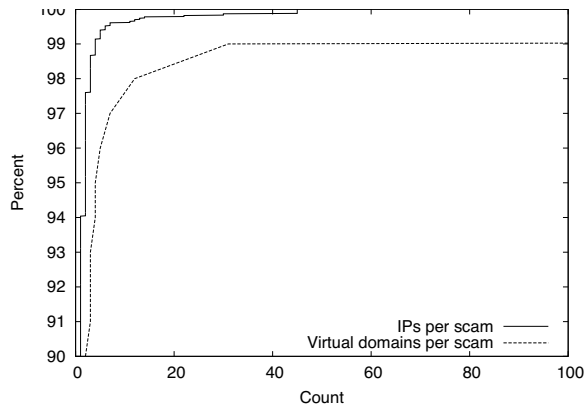


Figure 5: Number of IP address and virtual domains per scam.

36,390 unique URLs. Using image shingling, we identified 2,334 scams hosted on 7,029 machines. Spam is very redundant in advertising scams: on average, 100 spam messages with embedded URLs lead to only seven unique scams.

What kinds of scams do we observe in our trace? We use a commercial Web content filtering product to determine the prevalence of different kinds of scams. For every URL in our trace, we use the Web content filter to categorize the page downloaded from the URL. We then assign that category to the scams referenced by the URL.

Table 2 shows the ten most-prevalent scam categories. Note that we were not able to categorize all of the scams. We did not obtain access to the Web content filter until a few weeks after taking our traces, and 30% of the scams had URLs that timed out in DNS by that time (“Uncategorized” in the table). Further, 12% of the scams did not categorize due to the presence of dynamic content. The remaining 58% of scams fell into over 60 categories. Of these the most prevalent scam category was “Information Technology”, which, when examining the screenshots of the scam sites, include click affiliates, survey and free merchandise offers and some merchandise for sale (e.g., hair loss, software). Just over 2% of the scams were labeled as malicious sites (e.g., containing malware).

## 5.2 Distributed infrastructure

We start by evaluating to what extent scams use multiple hosts as distributed infrastructure. Scams might use multiple hosts for fault-tolerance, for resilience in anticipation of administrative takedown or blacklisting, for geographic distribution, or even for load balancing. Also, reports of large-scale botnets are increasingly common, and botnets could provide a large-scale infrastructure for hosting scams; do we see evidence of botnets being used as a scalable platform for scam hosting?

Scam category	# of domains	# of IPs
Watches	3029	3
Pharmacy	695	4
Watches	110	3
Pharmacy	106	1
Software	99	3
Male Enhancement	94	2
Phishing	91	14
Viagra	90	1
Watches	81	1
Software	80	45

Figure 6: The ten largest virtual-hosted scams and the number of IP addresses hosting the scams.

We count multiple scam hosting from two perspectives, the number of virtual hosts used by a scam and the number of unique IP addresses used by those virtual hosts. Overall, the scams from our trace are typically hosted on a single IP address with one domain name. Of the 2,334 scams, 2,195 (94%) were hosted on a single IP address and 1,960 (84%) were hosted on a single domain name. Only a small fraction of scams use multiple hosting. Figure 5 shows the tails of the distributions of the number of virtual hosts and IP addresses used by the scams in our trace, and Table 6 lists the top ten scams with the largest number of domains and IP addresses. Roughly 10% of the scams use three or more virtual domains, and 1% use 15 or more. The top scams use hundreds of virtual domains, with one scam using over 3,000. Of the 6% of scams hosted on multiple IP addresses, only a few used more than ten, with one scam using 45. The relatively prevalent use of virtual hosts suggests that scammers are likely concerned about URL blacklisting and use distinct virtual hosts in URLs sent in different spam messages to defeat such blacklists.

The scams in our trace do not use hosting infrastructure distributed across the network extensively. Most scams are hosted on a single IP address, providing a potentially convenient single point for network-based interdiction either via IP blacklisting or network filtering. Assuming that scammers adapt to defenses to remain effective, such filtering does not appear to be applied extensively. Scam serving workloads are apparently low enough that a single host can satisfy offered load sufficiently to reap the benefits of the scam. Finally, if scams do use botnets as hosting infrastructure, then they are not used to scale a single scam. A scammer could potentially use a botnet to host multiple different scams, hosting each scam on a separate distinct bot, but our methodology would not identify this case.

Those few scams hosted on multiple IP addresses, however, are highly distributed. Scams with multiple IP addresses were most commonly distributed outside of

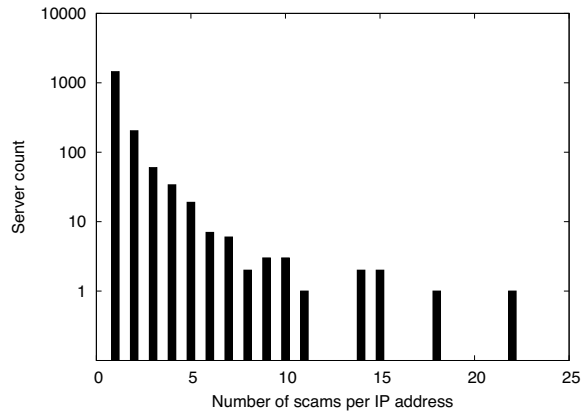


Figure 7: The number of scams found on a server IP address.

the same /24 prefix. Of the 139 distributed scams, all the hosts in 86% of the scams were located entirely on distinct /24 networks. Moreover, 64% of the distributed scams had host IP addresses that were all in entirely different ASes. As an example, one distributed scam was a phishing attack targeting a bank. The phishing Web pages were identical across 14 hosts, all in different /24 networks. The attack employed 91 distinct domain names. The domain names followed the same naming convention using a handful of common keywords followed by a set of numbers, suggesting the hosts were all involved in the distributed attack. The fully distributed nature of these scams suggests that scammers were concerned about resilience to defenses such as blacklisting.

### 5.3 Shared infrastructure

While we found that most scams are hosted on a single machine, a related question is whether these individual machines in turn host multiple scams, thereby sharing infrastructure across them. For each hosting IP address in our trace, we counted the number of unique scams hosted on that IP address at any time in the trace. Figure 7 shows these results as a logscale histogram. Shared infrastructure is rather prevalent: although 1,450 scams (62%) were hosted on their own machines, the remaining 38% of scams were hosted on machines hosting at least one other scam. Ten servers hosted ten or more scams, and the top three machines hosted 22, 18, and 15 different scams. This sharing of infrastructure suggests that scammers frequently either run multiple different scams on hosts that they control, or that hosts are made available (sold, rented, bartered) to multiple scammers.

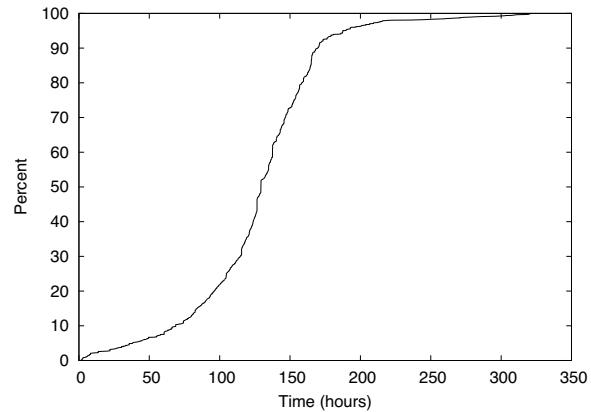


Figure 8: Overlap time for scam pairs on a server.

<i>Host type</i>	<i>Classification</i>	<i>% of hosts recognized</i>
Spam relay	Open proxy	72.3%
	Spam host	5.86%
Scam host	Open proxy	2.06%
	Spam host	14.9%

Table 3: Blacklist classification of spam relays and scam hosts.

#### 5.3.1 Sharing over time

We further examined these shared servers to determine if they host different scams sequentially or if, in fact, servers are used concurrently for different scams. For each pair of scams hosted on the same IP address, we compared their active times and durations with each other. When they overlapped, we calculated the duration of overlap. We found that scams sharing hosts shared them at the same time: 96% of all pairs of scams overlapped with each other when they remained active. Figure 8 shows the distribution of time for which scams overlapped. Over 50% of pairs of scams overlapped for at least 125 hours. Further calculating the ratio of time that scams sharing hosts were active, we found that overlapped scams did not necessarily start and end at the same time: only 10% of scam pairs fully overlapped each other.

#### 5.3.2 Sharing between scam hosts and spam relays

More broadly, how often do the same machines serve as both spam relays as well as scam hosting? Hosts used for both spam and scams suggest, for instance, that either the spammer and the scammer are the same party, or that a third party controls the infrastructure and makes it available for use by different clients. We can only estimate the extent to which hosts play both roles, but we

estimate it in two ways. First, we determine the IP addresses of all of the hosts that send spam into our feed. We then compare those addresses with the IP addresses of the scam hosts. Based upon this comparison, we find only a small amount of overlap (9.7%) between the scam hosts and spam relays in our trace.

Scam hosts could, of course, serve as spam relays that do not happen to send spam to our feed. For a more global perspective, we identify whether the spam and scam hosts we observe in our trace are blacklisted on well-known Internet blacklists. When the prober sees an IP address for the first time (either from a host sending spam or from a scam host), it performs a blacklist query on that IP address using the DNSBLLookup Perl module [16].

Table 3 shows the percentage of blacklisted spam relays and scam hosts. This perspective identifies a larger percentage (17%) of scam hosts as also sending spam than we found by comparing scam hosts and open relays within our spam feed, but the percentage is still small overall. The blacklists are quite effective, though, at classifying the hosts that send spam to our feed: 78% of those hosts are blacklisted. The query identifies most of the spam hosts as open spam relays — servers that forward mail and mask the identity of the true sender — whereas most blacklisted scam hosts are identified as just sending spam directly. These results suggest that when scam hosts are also used to send spam, they are rarely used as an open spam service.

## 5.4 Lifetime

Next we examine how long scams remain active and, in the next section, how stable they are while active. The lifetime of a scam is a balance of competing factors. Scammers have an incentive to use hosting infrastructure that provides longer lifetimes and higher availability to increase their rate of return. On the other hand, for example, numerous community and commercial services provide feeds and products to help network administrators identify, filter or take down some scam sites, particularly phishing scams [1, 6, 22, 25].

We define the lifetime of a scam as the time between the first and last successful timestamp for a probe operation during the two-week measurement period, independent of whether any probes failed in between (we look at the effect of failed probe attempts on availability below). We use two types of probes to examine scam host lifetime from different perspectives (Section 4). Periodic ping probes measure host network lifetime, and periodic HTTP requests measure scam server lifetime. Recall that we probe all hosts for a week after they appear in our spam feed — and no longer — to remove any bias towards hosts that appear early in the measurement study.

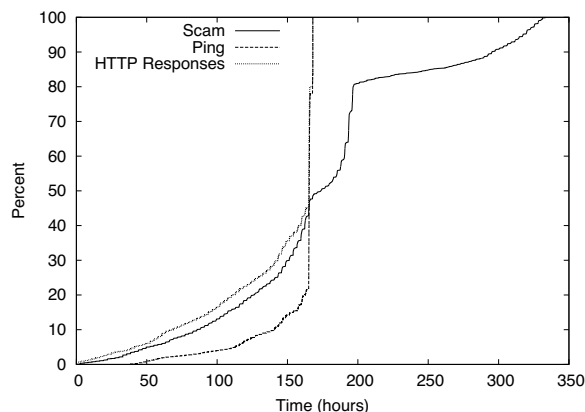


Figure 9: Lifetimes of individual scam hosts and Web servers, as well overall lifetimes of scams across multiple hosts.

For comparison, we also calculate the lifetimes of entire scams. For scams that use multiple hosts, their lifetimes start when the first host appears in our trace and end with the lifetime of the last host to respond. As a result, scam lifetimes can exceed a week.

How long are scams active? Figure 9 shows the distributions of scam lifetime based upon these probes for the scams in our trace. For ping probes, we show the distribution of just those scam hosts that responded to pings (67% of all scam hosts). Scam hosts had long network lifetimes. Over 50% of hosts responded to pings for nearly the entire week that we probed them, and fewer than 10% of hosts responded to pings for less than 80 hours. Given how close the distributions are, scam Web servers had only slightly shorter lifetimes overall. These results suggest that scam hosts are taken down soon after scam servers.

Comparing the distribution of scam lifetimes to the others, we see that scams benefit from using multiple hosts. The 50% of scams whose lifetimes exceed a week indicate that the lifetimes of the individual scam hosts do not entirely overlap each other. Indeed, individual hosts for some scams appeared throughout the week of our measurement study, and the overall scam lifetime approached the two weeks.

### 5.4.1 Lifetime by category

A substantial amount of community and commercial effort goes into identifying malicious sites, such as phishing scams, and placing those sites on URL or DNS/IP blacklists. Thus, we would expect that the hosting infrastructure for clearly malicious scams would be more transient than for other scams. To test this hypothesis, we used the categorization of scams to create a group of malicious scams that include the “Illegal or Question-

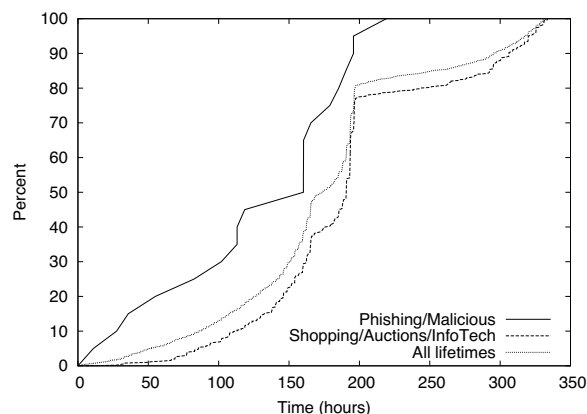


Figure 10: Scam lifetime distributions for malicious and shopping scams.

able” and “Phishing” categories labeled by the Web content filter (32 scams). For comparison, we also broke out another group of more innocuous shopping scams that include the “Shopping”, “Information Technology”, and “Auction” categories (701 scams).

We examined the lifetimes and prevalence on blacklists of these scams. Figure 10 shows the lifetime distributions of the malicious and shopping groups of scams, and includes the distribution of all scams from Figure 9 for reference. The malicious scams have a noticeably shorter lifetime than the entire population, and the shopping scams have a slightly longer lifetime. Over 40% of the malicious scams persist for less than 120 hours, whereas the lifetime for the same percentage of shopping scams was 180 hours and the median for all scams was 155 hours. These results are consistent with malicious scam sites being identified and taken down faster than other scam sites, although we cannot verify the causality.

As further evidence, we also examined the prevalence of malicious scams on the DNS blacklists we use in Section 5.3.2, and compare it to the blacklisting prevalence of all scams and the shopping scams. Over 28% of the malicious scams were blacklisted, roughly twice as often as the shopping scams (12% blacklisted) and all scams (15%). Again, these results are consistent with the lifetimes of malicious scams — being blacklisted twice as frequently could directly result in shorter scam lifetimes.

#### 5.4.2 Spam campaign lifetime

A related aspect to scam lifetime are the “spam campaigns” used to advertise scams and attract clients. We captured 319,700 spam emails with links in our trace, resulting in 2,334 scams; on average, then, each scam was advertised by 137 spam emails. We use these repeated spam emails to determine the lifetime of spam

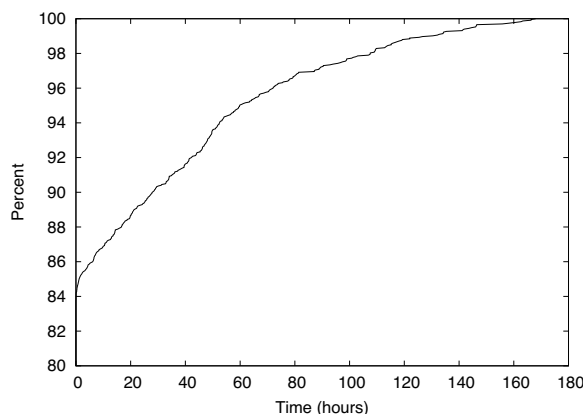


Figure 11: The duration of a spam campaign.

campaigns for a scam by measuring the time between the first and last spam email messages advertising that scam. Figure 11 shows the distribution of the spam campaign lifetimes. Compared to the lifetime of scam sites, most spam campaigns are relatively short. Over 50% of the campaigns last less than 12 hours, over 90% last less than 48 hours, and 99% last less than three days. Roughly speaking, the lifecycle of a typical scam starts with a short spam campaign lasting half of a day while the scam site remains up for at least a week.

The relative lifetimes of spam campaigns and scam hosts again reflect the different needs of the two services. Compared with scam hosts, spam relays need to be active for much shorter periods of time to accomplish their goals. Spammers need only a window of time to distribute spam globally; once sent, spam relays are no longer needed for that particular scam. Scam hosts, in contrast, need to be responsive and available for longer periods of time to net potential clients. Put another way, spam is blanket advertising that requires no interaction with users to deliver, whereas scam hosting is a service that fundamentally depends upon user interaction to be successful. In contrast, scam hosts benefit more from stable infrastructure that remains useful and available for much longer periods of time.

### 5.5 Stability

A profitable scam requires stable infrastructure to serve potential customers at any time, and for as long as the scam is active. To gauge the stability of scam hosting infrastructure, we probed each scam host periodically for a week to measure its availability. When downloading pages from the hosts, we also used p0f to fingerprint host operating systems and link connectivity.

We computed scam availability as the number of successful Web page downloads divided by the total number

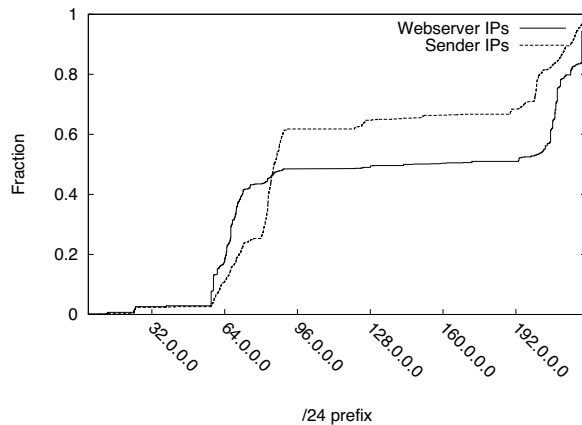


Figure 12: IP addresses, binned by /24 prefix, for spam sending relays and scam host servers.

of download attempts within the overall lifetime of the scam; if a scam lasted for only three days, we computed availability only during those days. Scams had excellent availability: over 90% of scams had an availability of 99% or higher. Of the remaining, most had availabilities of 98% or higher. As fingerprinted by p0f, more scams ran on Unix or server appliances (43%) than Windows systems (30%), and all of them had reported good link connectivity. These results indicate that scam hosting is quite reliable within the lifetime of a scam.

## 5.6 Scam location

We next examine both the network and geographic locations of scam hosts. For comparison, we also examine the locations of the spam relays that sent the spam in our trace. Comparing them highlights the extent to which the different requirements of the two services reflect where around the world and in the network they are found.

### 5.6.1 Network location

The network locations of spam relays and scam hosts are more consistent. Figure 12 shows the cumulative distribution of IP addresses for spam relays and scam hosts in our trace. Consistent with a similar analysis of spam relays in [27], the distributions are highly non-uniform. The IP addresses of most spam relays and scam hosts fall into the two same ranges, 58.\* to 91.\* and 200.\* to 222.\*. However, within those two address ranges hosts for the two services have different concentrations. The majority of spam relays (over 60%) fall into the first address range and are distributed somewhat evenly except for a gap between 70.\* and 80.\*. Roughly half of the scam hosts also fall into the first address range, but most of those

<i>Scam host country</i>	<i>% of all servers</i>
United States	57.40%
China	7.23%
Canada	3.70%
Great Britain	3.07%
France	3.06%
Germany	2.52%
Russia	1.80%
South Korea	1.77%
Japan	1.60%
Taiwan	1.53%
Other	16.32%

Table 4: Countries of scam hosts.

<i>Spam relay country</i>	<i>% of all relays</i>
United States	14.50%
France	7.06%
Spain	6.75%
China	6.65%
Poland	5.68%
India	5.42%
Germany	5.00%
South Korea	4.67%
Italy	4.44%
Brazil	3.86%
Other	30.97%

Table 5: Countries of spam relays.

fall into the 64.\* to 72.\* subrange and relatively few in the second half of the range. Similarly, scams are more uniformly distributed within the second address range as well.

### 5.6.2 Geographic location

How do these variations in network address concentrations map into geographic locations? The effectiveness of scams could relate to (at least perceived) geographic location. As one anecdote, online pharmaceutical vendors utilized hosting servers inside the United States to imply to their customers that they were providing a lawful service [24].

Using Digital Element's NetAcuity tool [10], we mapped the IP addresses of scam hosts to latitude and longitude coordinates. Using these coordinates, we then identified the country in which the host was geographically located. Table 4 shows the top ten countries containing scam hosts in our trace. Interestingly, the NetAcuity service reported that nearly 60% of the scam hosts are located in the United States. Overall, 14% were located in Western Europe and 13% in Asia. For compar-

ison, Table 5 shows the top ten countries containing spam relays. The geographic distributions for spam relays are quite different than scam hosts. Only 14% of spam relays are located in the United States, whereas 28% are located in Western Europe and 16% in Asia. We also found the top ASes for scam hosts and senders, but found no discernible pattern and omit the results for brevity.

The strong bias of locating scam hosts in the United States suggests that geographic location is more important to scammers than spammers. There are a number of possible reasons for this bias. One is the issue of perceived enhanced credibility by scammers mentioned above. Another relates to the difference in requirements for the two types of services. As discussed in Section 5.4.2, spam relays can take advantage of hosts with much shorter lifetimes than scam hosts. As a result, spam relays are perhaps more naturally suited to being hosted on compromised machines such as botnets; the compromised machine need only be under control of the spammer long enough to launch the spam campaign. Scam hosts benefit more from stability, and hosts and networks within the United States can provide this stability.

## 6 Conclusion

This paper does not study spam itself, nor the infrastructure used to deliver spam, but rather focuses on the scam infrastructure that is nourished by spam. We demonstrate the *spamscatter* technique for identifying scam infrastructure and how to use approximate image comparison to cluster servers according to individual scams — side-stepping the extensive content and networking camouflaging used by spammers.

From a week-long trace of a large real-time spam feed (roughly 150,000 per day), we used the *spamscatter* technique to identify and analyze over 2,000 distinct scams hosted across more than 7,000 distinct servers. We found that, although large numbers of hosts are used to advertise Internet scams using spam campaigns, individual scams themselves are typically hosted on only one machine. Further, individual machines are commonly used to host multiple scams, and occasionally serve as spam relays as well. This practice provides a potentially convenient single point for network-based interdiction either via IP blacklisting or network filtering.

The lifecycle of a typical scam starts with a short spam campaign lasting half of a day while the scam site remains up for at least a week. The relative lifetimes of spam campaigns and scam hosts reflect the different requirements of the two underground services. Spam is blanket advertising that requires no interaction with users to deliver, whereas scam hosting is a service that fundamentally depends upon user interaction to be successful. Finally, mapping the geographic locations of scam hosts,

we found that they have a strong bias to being located in the United States. The strong bias suggests that geographic location is more important to scammers than spammers, perhaps due to the stability of hosts and networks within the U.S.

## Acknowledgments

We would like to thank a number of people who made contributions to this project. We are particularly grateful to Weidong Cui and Christian Kreibich, who maintained the spam feed we used for our analyses, the anonymous party who gave us access to the spam feed itself, and Vern Paxson for discussions and feedback. Kirill Levchenko suggested image-based comparison of Web pages as an equivalence test, and Colleen Shannon assisted us with Digital Element's NetAcuity tool. Finally, we would like to also thank the anonymous reviewers for their comments, the CCIED group for useful feedback on the project, and Chris X. Edwards for system support. Support for this work was provided in part by NSF under CyberTrust Grant No. CNS-0433668 and AFOSR MURI Contract F49620-02-1-0233.

## References

- [1] ANTI-PHISHING WORKING GROUP. Report Phishing. <http://www.antiphishing.org/>.
- [2] BARFORD, P., BESTAVROS, A., BYERS, J., AND CROVELLA, M. On the marginal utility of network topology measurements. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop* (Oct. 2001).
- [3] BÖHME, R., AND HOLZ, T. The effect of stock spam on financial markets. In *Proceedings of the Fifth Workshop on the Economics of Information Security (WEIS 2006)* (June 2006).
- [4] BRODER, A. Z. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences (SEQUENCES'97)* (June 1997), pp. 21–29.
- [5] CASADO, M., GARFINKEL, T., CUI, W., PAXSON, V., AND SAVAGE, S. Opportunistic measurement: Extracting insight from spurious traffic. In *Proceedings of the 4th ACM Workshop on Hot Topics in Networks (HotNets-IV)* (College Park, MD, Nov. 2005).
- [6] CASTLECOPS. Fried Phish: Phishing Incident Reporting and Termination (PIRT). <http://www.calecop.com/pirt>.
- [7] CHOU, N., LEDESMA, R., TERAGUCHI, Y., BONEH, D., AND MITCHELL, J. C. Client-side defense against Web-based identity theft. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)* (Feb. 2004).
- [8] COOKE, E., BAILEY, M., MAO, Z. M., WATSON, D., JAHANIAN, F., AND MCPHERSON, D. Toward understanding distributed blackhole placement. In *Workshop on Rapid Malcode (WORM'04)* (Oct. 2004).
- [9] COOKE, E., JAHANIAN, F., AND MCPHERSON, D. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Proceedings of the First Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI'05)* (July 2005).

- [10] DIGITAL ELEMENT. NetAcuity IP Intelligence. [http://www.digital-element.net/ip\\_intelligence/ip\\_intelligence.html/](http://www.digital-element.net/ip_intelligence/ip_intelligence.html/).
- [11] FETTERLY, D., MANASSE, M., AND NAJORK, M. On the evolution of clusters of near-duplicate Web pages. In *Proceedings of the First Latin American Web Congress* (Nov. 2003), pp. 37–45.
- [12] GILLIS, T. Internet Security Trends for 2007. Ironport Whitepaper, 2007.
- [13] IRONPORT INC. Spammers continue innovation. IronPort press release, June 28, 2006. [http://www.ironport.com/company/ironport\\_pr\\_...html](http://www.ironport.com/company/ironport_pr_...html).
- [14] KDE. Khtml layout engine. <http://www.kde.org/>.
- [15] KEIZER, G. Spam volume jumps 35% in November, Dec. 2006. [http://informationweek.com/news/how\\_rtcicle.jhtml\\_article](http://informationweek.com/news/how_rtcicle.jhtml_article).
- [16] MATHER, T. Net::DNSBLLookup Perl module. <http://search.cpan.org/tjmather/etooop/>.
- [17] MESSAGELABS. 2006: The year spam raised its game and threats got personal, Dec. 2006. [http://www.messagelab.com/publichedcontent/public\\_h/about\\_u\\_dotcom\\_en/new\\_\\_\\_event/pre\\_relea\\_e/\\_...html](http://www.messagelab.com/publichedcontent/public_h/about_u_dotcom_en/new___event/pre_relea_e/_...html).
- [18] MONGA, V., AND EVANS, B. L. Robust perceptual image hashing using feature points. In *Proceedings of the IEEE International Conference on Image Processing (ICIP'04)* (Oct. 2004), pp. 677–680.
- [19] MOORE, D., PAXSON, V., SAVAGE, S., SHANNON, C., STANFORD, S., AND WEAVER, N. Inside the Slammer worm. *IEEE Security and Privacy* 1, 4 (July 2003), 33–39.
- [20] MOORE, D., SHANNON, C., BROWN, D., VOELKER, G. M., AND SAVAGE, S. Inferring Internet denial-of-service activity. *ACM Transactions on Computer Systems* 24, 2 (May 2006), 115–139.
- [21] MOORE, D., SHANNON, C., AND BROWN, J. Code-Red: a case study on the spread and victims of an Internet worm. In *Proceedings of the ACM/USENIX Internet Measurement Workshop (IMW)* (Marseille, France, Nov. 2002).
- [22] NEWS.ADMIN.NET-ABUSE.SIGHTINGS. USENET newsgroup for discussion of spam. <http://www.nanae.org/>.
- [23] PAUL BÄHER, THORSTEN HOLZ, MARKUS KÖTTER AND GEORG WICHESKI. Know your enemy: Tracking botnets. In *The Honeynet Project & Research Alliance* (Mar. 2005).
- [24] PHILADELPHIA INQUIRER. Special reports: Drugnet. [http://www.philly.com/mld/inquirer/new/pecial\\_pac\\_age/pill/](http://www.philly.com/mld/inquirer/new/pecial_pac_age/pill/).
- [25] PHISHTANK. Join the fight against phishing. <http://www.phishtank.com/>.
- [26] RAJAB, M. A., ZARFOSS, J., MONROSE, F., AND TERZIS, A. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the ACM Internet Measurement Conference* (Rio de Janeiro, Brazil, Oct. 2006).
- [27] RAMACHANDRAN, A., AND FEAMSTER, N. Understanding the network-level behavior of spammers. In *Proceedings of the ACM SIGCOMM Conference* (Pisa, Italy, Sept. 2006).
- [28] ROOVER, C. D., VLEESCHOUWER, C. D., LEFEBVRE, F., AND MACQ, B. Robust image hashing based on radial variance of pixels. In *Proceedings of the IEEE International Conference on Image Processing (ICIP'05)* (Sept. 2005), pp. 77–80.
- [29] SHIVAKUMAR, N., AND GARCIA-MOLINA, H. Finding near-replicas of documents and servers on the Web. In *Proceedings of the First International Workshop on the Web and Databases (WebDB'98)* (Mar. 1998).
- [30] VENKATESAN, R., KOON, S. M., JAKUBOWSKI, M. H., AND MOULIN, P. Robust image hashing. In *Proceedings of the IEEE International Conference on Image Processing (ICIP'00)* (Sept. 2000).
- [31] WEBB, S., CAVERLEE, J., AND PU, C. Introducing the Webb spam corpus: Using email spam to identify Web spam automatically. In *Proceedings of the 3rd Conference on Email and Anti-Spam (CEAS)* (Mountain View, 2006).
- [32] YEGNESWARAN, V., BARFORD, P., AND PLONKA, D. On the design and use of Internet sinks for network abuse monitoring. In *Proceedings of Recent Advances on Intrusion Detection* (Sept. 2004).

# Exploiting Network Structure for Proactive Spam Mitigation

Shobha Venkataraman<sup>\*</sup>, Subhabrata Sen<sup>†</sup>, Oliver Spatscheck<sup>†</sup>, Patrick Haffner<sup>†</sup>, Dawn Song<sup>\*</sup>

<sup>\*</sup>Carnegie Mellon University, <sup>†</sup>AT&T Research

shobha@cs.cmu.edu, {sen,spatsch,haffner}@research.att.com, dawnsong@cmu.edu

## Abstract

E-mail has become indispensable in today's networked society. However, the huge and ever-growing volume of spam has become a serious threat to this important communication medium. It not only affects e-mail recipients, but also causes a significant overload to mail servers which handle the e-mail transmission.

We perform an extensive analysis of IP addresses and IP aggregates given by network-aware clusters in order to investigate properties that can distinguish the bulk of the legitimate mail and spam. Our analysis indicates that the bulk of the legitimate mail comes from long-lived IP addresses. We also find that the bulk of the spam comes from network clusters that are relatively long-lived. Our analysis suggests that network-aware clusters may provide a good aggregation scheme for exploiting the history and structure of IP addresses.

We then consider the implications of this analysis for prioritizing legitimate mail. We focus on the situation when mail server is overloaded, and the goal is to maximize the legitimate mail that it accepts. We demonstrate that the history and the structure of the IP addresses can reduce the adverse impact of mail server overload, by increasing the number of legitimate e-mails accepted by a factor of 3.

## 1 Introduction

E-mail has emerged as an indispensable and ubiquitous means of communication today. Unfortunately, the ever-growing volume of spam diminishes the efficiency of e-mail, and requires both mail server and human resources to handle.

Great effort has focused on reducing the amount of spam that the end-users receive. Most Internet Service Providers (ISPs) operate various types of spam filters [1, 4, 5, 13] to identify and remove spam e-mails before they are received by the end-user. E-mail software on end-hosts adds an additional layer of filtering to remove this

unwanted traffic, based on the typical email patterns of the end-user.

Much less attention has been paid to how the large volume of spam impacts the mail infrastructure within an ISP, which has to receive, filter and deliver them appropriately. Spammers have a strong incentive to send large volumes of spam – the more spam they send, the more likely it is that some of it can evade the spam filters deployed by the ISPs. It is easy for the spammer to achieve this – by sending spam using large botnets, spammers can easily generate far more messages than even the largest mail servers can receive. In such conditions, it is critical to understand how the mail server infrastructure can be made to prioritize legitimate mail, processing it preferentially over spam.

In this context, the requirements for differentiating between spam and non-spam are slightly different from regular spam-filtering. The primary requirement for regular spam-filtering is to be conservative in discarding spam, and for this, computational cost is not usually a consideration. However, when the mail server must prioritize the processing of legitimate mail, it has to use a computationally-efficient technique to do so. In addition, in this situation, even an imperfect distinction criterion would be useful, as long as a significant fraction of the legitimate mail gets classified correctly.

In this paper, we explore the potential of using the historical behaviour of IP addresses to predict whether an incoming email is likely to be legitimate or spam. Using IP addresses for classification is computationally substantially more efficient than any content-based techniques. IP address information can also be collected easily and is more difficult for a spammer to obfuscate. Our measurement studies show that IP address information provides a stable discriminator between legitimate mail and spam. We find that good mail servers send mostly legitimate mail and are persistent for significant periods of time. We also find that the bulk of spam comes from IP prefixes that send mostly spam and are also persis-

tent. With these two findings, we can use the properties of *both* legitimate mail and spam together, rather than using the properties of only legitimate mail or only spam, in order to prioritize legitimate mail when needed.

We show that these measurements are valuable in an application where legitimate mail must be prioritized. We focus on the situation when mail servers are overloaded, i.e., they receive far more mail than they can process, even though the legitimate mail received is a tiny fraction of the total received. Since mail typically gets dropped at random when the server is overloaded, and spam can be generated at will, the spammer has an incentive to overload the server. Indeed, the optimal strategy for the spammer is to increase the load on the mail infrastructure to a point where the most spam will be accepted by the server; this kind of behaviour has been observed on the mail servers of large ISPs. In this paper, we show an application of our measurement study to design techniques based on the reputations of IP addresses and their aggregates and demonstrate the benefits to the mail server overload problem.

The contributions of this paper are two-fold. We first perform an extensive measurement study in order to understand some IP-based properties of legitimate mail and spam. We then perform a simulation study to evaluate how we can use these properties to prioritize legitimate mail when the mail server is overloaded.

Our main results are the following:

- We find that a significant fraction of legitimate mail comes from IP addresses that last for a long time, even though a very significant fraction of spam comes from IP addresses that are ephemeral. This suggests that the history of “good” IP addresses, that is, IP addresses that send mostly legitimate mail, could be used for prioritizing mail in spam mitigation.
- We explore *network-aware clusters* as a candidate aggregation scheme to exploit structure in IP addresses. Our results suggest that IP addresses responsible for the bulk of the spam are well-clustered, and that the clusters responsible for the bulk of the spam are persistent. This suggests that network-aware clusters may be good candidates to assign reputations to unknown IP addresses.
- Based on our measurement results, we develop a simple reputation scheme that can prioritize IP addresses when the server is overloaded. Our simulations show that when the server receives many more connection requests than it can process, our policy gives a factor of 3 improvement in the number of legitimate mails accepted.

We note that the server overload problem is just one application that illustrates how IP information could be used for prioritizing email. This information could be used to prioritize e-mail at additional points of the mail server infrastructure as well. However, the kind of structural information that is reflected in the IP addresses may not always be a perfect discriminator between spammers and senders of legitimate mail, and this is, indeed, reflected in the measurements. Such structural IP information could, therefore, be used in combination with other techniques in a general-purpose spam mitigation system, and this information is likely to be useful by itself only when an aggressive and computationally-efficient technique is needed.

The remainder of the paper is structured as follows. We present our analysis of characteristics of IP addresses and network-aware clusters that distinguish between legitimate mail and spam in Sections 2 and 3 respectively. We present and evaluate our solution for protecting mail servers under overload in Section 4. We review related work in Section 5 and conclude in Section 6.

## 2 Analysis of IP-Address Characteristics

In this section, we explore the extent to which IP-based identification can be used to distinguish spammers from senders of legitimate e-mail based on differences in patterns of behaviour.

### 2.1 Data

Our data consists of traces from the mail server of a large company serving one of its corporate locations with approximately 700 mailboxes, taken over a period of 166 days from January to June 2006. The location runs a PostFix mail server with extensive logging that records the following: (a) every attempted SMTP connection, with its IP address and time stamp, (b) whether the connection was rejected, along with a reason for rejection, (c) if the connection was accepted, results of additional mail server’s local spam-filtering tests, and if accepted for delivery, the results of running SpamAssassin.

Fig. 1(a) shows a daily summary of the data for six months. It shows four quantities for each day: (a) the number of SMTP connection requests made (including those that are denied via blacklists), (b) the number of e-mails received by the mail server, (c) the number of e-mails that were sent to SpamAssassin, and (d) the number of e-mails deemed legitimate by SpamAssassin. The relative sizes of these four quantities on every day illustrate the scale of the problem: spam is 20 times larger than the legitimate mail received. (In our data set, there were 1.4 million legitimate messages and 27 million spam messages in total.) Such a sharp imbalance

indicates the potential of a significant role for applications like maximizing legitimate mail accepted when the server is overloaded: if there is a way to prioritize legitimate mail, the server could handle it much more quickly, because the volume of legitimate mail is tiny in comparison to spam.

In the following analysis, every message that is considered legitimate by SpamAssassin is counted as a legitimate message; every message that is considered spam by SpamAssassin, the mail server's local spam-filtering tests, or through denial by a blacklist is counted as spam.

## 2.2 Analysis of IP Addresses

We first explore the behaviour of individual IP addresses that send legitimate mail and spam, with the goal of uncovering any significant differences in their behavioral patterns.

Our analysis focuses on the *IP spam-ratio* of an IP address, which we define to be the fraction of mail sent by the IP address that is spam. This is a simple, intuitive metric that captures the spamming behaviour of an IP address: a low spam-ratio indicates that the IP address sends mostly legitimate mail; a high spam-ratio indicates that the IP address sends mostly spam. Our goal is to see whether the historical communication behaviour of IP addresses categorized by their spam-ratios can differentiate between IP addresses of legitimate senders and spammers, for spam mitigation.

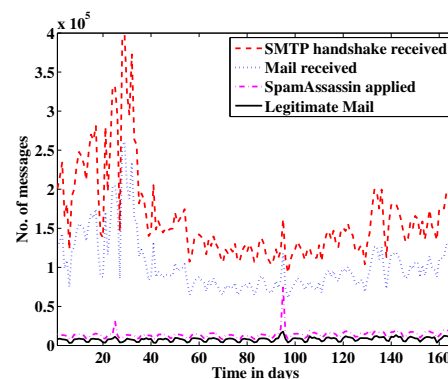
As discussed earlier, the differentiation between the legitimate senders and spammers need not be perfect; there are benefits to having even a partial differentiation, especially with a simple, computationally inexpensive feature. For example, in the server overload problem, when all the mail cannot be accepted, a partial separation would still help to increase the amount of legitimate mail that is received.

In the IP-based analysis, we will address the following questions:

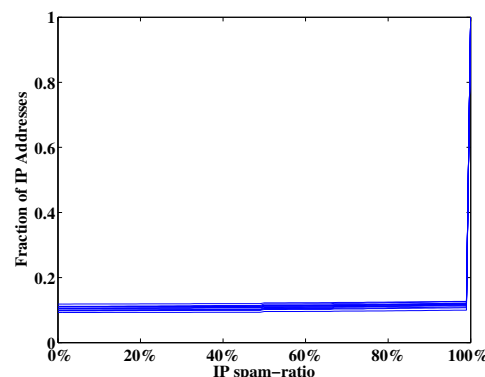
- *Distribution by IP Spam Ratio*: What is the distribution of IP addresses by their spam-ratio, and what fraction of legitimate mail and spam is contributed by IP addresses with different spam-ratios?
- *Persistence*: Are IP addresses with low/high spam-ratios present across long time periods? If they are, do such IP addresses contribute to a significant fraction of the legitimate mail/spam?
- *Temporal Spam-Ratio Stability*: Do many of the IP addresses that appear to be good on average fluctuate between having very low and very high spam-ratios?

The answers to these three questions, taken together, give us an idea of the benefit we could derive in using the history of IP address behaviour in spam mitigation. We show in Sec. 2.2.1, that most IP addresses have a spam-ratio of 0% or 100%, and also that a significant amount of the legitimate mail comes from IP addresses whose spam-ratio exceeds zero. In Sec. 2.2.2, we show that a very significant fraction of the legitimate mail comes from IP addresses that persist for a long time, but only a tiny fraction of the spam comes from IP addresses that persist for a long time. In Sec. 2.2.3, we show that most IP addresses have a very high temporal ratio-stability – they do not fluctuate between exhibiting a very low or very high daily spam-ratio over time.

Together, these three observations suggest that *identifying IP addresses with low spam ratios that regularly send legitimate mail* could be useful in spam mitigation and prioritizing legitimate mail. In the rest of this section, we present the analysis that leads to these observations. For concreteness, we focus on how the analysis can help spam mitigation in the server overload problem.



(a) Data characteristics



(b) CDFs of IP spam-ratios for many days: each line is a CDF for a different day.

Figure 1: 1(a): Daily summary of the data set over 6 months. 1(b): CDFs of IP spam-ratios for many different days.

### 2.2.1 Distribution by IP Spam-Ratio

In this section, we explore how the IP addresses and their associated mail volumes are distributed as a function of the IP spam-ratios. We focus here on the spam-ratio computed over a short time period in order to understand the behaviour of IP addresses without being affected by their possible fluctuations in time. Effectively, this analysis shows the limits of the differentiation that could be achieved by using IP spam-ratio, even assuming that IP spam-ratio could be predicted for a given IP address over short periods of time. In this section, we focus on day-long intervals, in order to take into account possible time-of-day variations. We refer to the IP spam-ratio computed over a day-long interval as the *daily spam-ratio*.

Intuitively, we expect that most IP addresses either send mostly legitimate mail, or mostly spam, and that most of the legitimate mail and spam comes from these IP addresses. If this hypothesis holds, then for spam mitigation, it will be sufficient if we can identify the IP addresses as senders of legitimate mail or spammers. To test this hypothesis, we analyze the following two empirical distributions: (a) the distribution of IP addresses as a function of the spam-ratios, and (b) the distribution of legitimate mail/spam as a function of their respective IP addresses' spam-ratio.

We first analyze the distribution of IP addresses by their daily spam-ratios in Fig. 1(b). For each day, it shows the empirical cumulative distribution function (CDF) of the daily spam-ratios of individual IP addresses active on that day. Fig. 1(b) shows this daily CDF for a large number of randomly selected days across the observation period.

**Result 1. Distribution of IP addresses:** (i) *Most IP addresses, send either mostly spam or mostly legitimate mail.* (ii) *Fewer than 1 – 2% of the active IP addresses have a spam-ratio of between 1% – 99%, i.e., there are very few IP addresses that send a non-trivial fraction of both spam and legitimate mail.* (iii) *Further, the vast majority (nearly 90%) of IP addresses on any given day generate almost exclusively spam, and have spam-ratios between 99% – 100%.*

The above results indicate that identifying IP addresses with low or high spam-ratios could identify most of the legitimate senders and spammers. In addition, for some applications (e.g., the mail server overload problem), it would be valuable to identify the IP addresses that send the bulk of the spam or the bulk of the legitimate mail, in terms of mail volume. To do so, we next explore how the daily legitimate mail or spam volumes are distributed as a function of the IP spam-ratios, and the resulting implications.

Let  $I_k$  denote the set of all IP addresses that have a spam-ratio of at most  $k$ . Fig. 2 examines how the volume

of legitimate mail and spam sent by the set  $I_k$  depends on the spam-ratio  $k$ . Specifically, let  $L_i(k)$  and  $S_i(k)$  be the fractions of the total daily legitimate mail and spam that comes from all IPs in the set  $I_k$ , on day  $i$ . Fig. 2(a) plots  $L_i(k)$  averaged over all the days, along with confidence intervals. Fig. 2(b) shows the corresponding distribution for the spam volume  $S_i(k)$ .

**Result 2. Distribution of legitimate mail volume:** *Fig. 2(a) shows that the bulk of the legitimate mail (nearly 70% on average) comes from IP addresses with a very low spam-ratio ( $k \leq 5\%$ ). However, a modest fraction (over 7% on average) also comes from IP addresses with a high spam-ratio ( $k \geq 80\%$ ).*

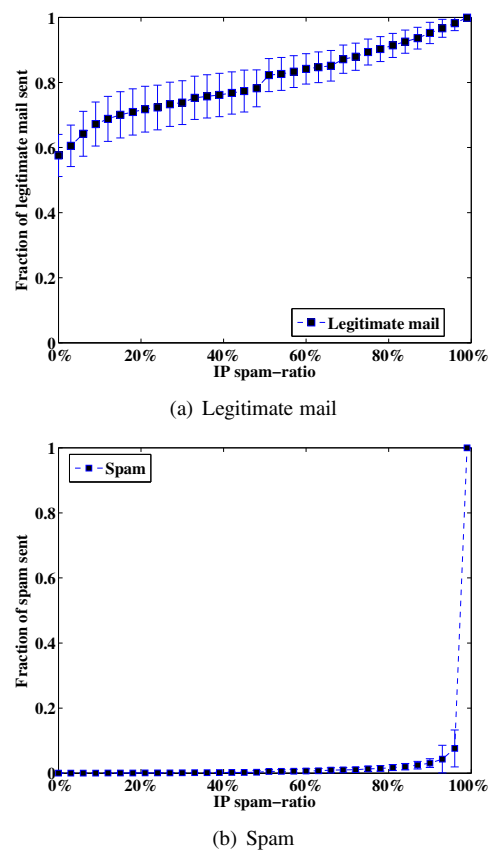


Figure 2: Legitimate mail and spam contributions as a function of IP spam-ratio.

**Result 3. Distribution of spam volume:** *Fig. 2(b) indicates that almost all (over 99% on average) of the spam sent every day comes from IP addresses with an extremely high spam-ratio (when  $k \geq 95\%$ ). Indeed, the contribution of the IP addresses with lower spam-ratios ( $k \leq 80\%$ ) is a tiny fraction of the total.*

We observe that the distribution of legitimate mail volume as a function of the spam-ratio  $k$  is more diffused

than the distribution of spam volume. There are two possible explanations for such behaviour of the legitimate senders. First, spam-filtering software tends to be conservative, allowing some spam to be marked as legitimate mail. Second, a lot of legitimate mail tends to come from large mail servers that cannot do perfect outgoing spam-filtering. These mail servers may, therefore, have a slightly higher IP spam-ratio, and this would cause the distribution of legitimate mail to be more diffused across the spam-ratio.

Together, the above results suggest that the IP spam-ratio may be a useful discriminating feature for spam mitigation. As an example, assume that we have a classification function that accepted (or prioritized) all IP addresses with a spam-ratio of at most  $k$  and rejected all IP addresses with a higher spam-ratio. Then, if we set  $k = 95\%$ , we could accept (or prioritize) nearly all the legitimate mail, and no more than 1% of the spam. However, such a classification function requires perfect knowledge of every IP address's daily spam-ratio every single day, and in reality, this knowledge may not be available.

Instead, our approach is to identify properties that occur over longer periods of time, and are useful for predicting the current behaviour of an IP address based on long-term history, and these properties are incorporated into classification functions. The effectiveness of such history-based classification functions for spam mitigation depends on the extent to which IP addresses long-lived, how much of the legitimate email or spam are contributed by the long-lived IP addresses, and to what extent the spam-ratio of an IP address varies over time. Sec. 2.2.2 and Sec. 2.2.3 explore these questions.

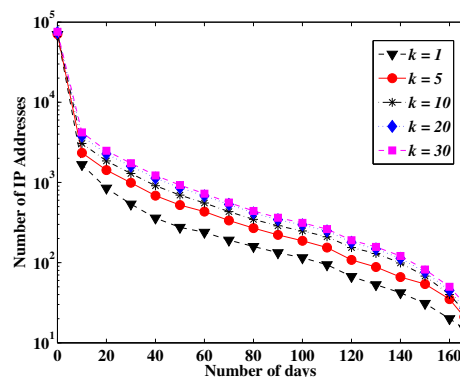
For the following analysis, we focus on the spam-ratio of each individual IP address, computed over the entire data set, since we are interested in its behaviour over its lifetime. We refer to this as the *lifetime spam-ratio* of the IP address. We show the presence of two properties in this analysis: (i) a significant fraction of legitimate mail comes from good IP addresses that last for a long time (*persistence*), and (ii) IP addresses that are good on average tend to have a low spam-ratio each time they appear (*temporal stability*). These two properties directly influence how effective it would be to use historical information for determining the likelihood of spam coming from an individual IP address.

## 2.2.2 Persistence

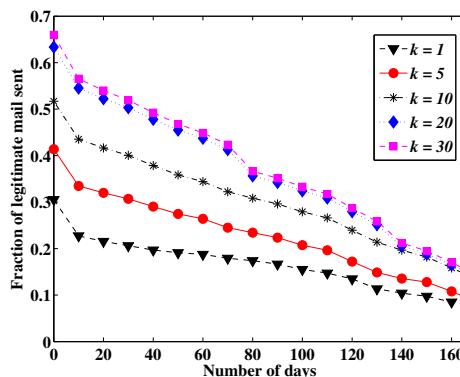
Due to the community structure inherent in non-spam communication patterns, it seems reasonable that most of the legitimate mail will originate from IP addresses that appear and re-appear. Previous studies have also indicated that most of the spam comes from IP addresses that

are extremely short-lived. These suggest the existence of a potentially significant difference in the behaviour of senders of legitimate mail and spammers with respect to persistence. We next quantify the extent to which these hypotheses hold, by examining the persistence of individual IP addresses.

Our methodology for understanding the persistence behavior of IP addresses is as follows: we consider the set of all IP addresses with a low lifetime spam-ratio and examine how much legitimate mail they send, as well as how much of the legitimate mail is sent by IP addresses that are present for a long time. Such an understanding can indicate the potential of using a whitelist-based approach for prioritizing legitimate mail. If, for instance, the bulk of the legitimate mail comes from IP addresses that last for a long time, we could use this property to prioritize legitimate mail from long-lasting IP addresses with low spam-ratios.



(a) Number of  $k$ -good IP addresses present for  $x$  or more days



(b) Fraction of legitimate mail sent by  $k$ -good IP addresses present for  $x$  or more days

Figure 3: Persistence of  $k$ -good IP addresses.

For this analysis, we use the following two definitions.

**Definition 1.** A  $k$ -good IP address is an IP address whose lifetime spam-ratio is at most  $k$ . A  $k$ -good set is the set of all  $k$ -good IP addresses. Thus, a 20-good set

is the set of all IP addresses whose lifetime spam-ratio is no more than 20%.

We compute (a) the number of  $k$ -good IP addresses present for at least  $x$  distinct days, and (b) the fraction of legitimate mail contributed by  $k$ -good IP addresses that are present in at least  $x$  distinct days.<sup>1</sup> Fig. 3(a) shows the number of IP addresses that appear in at least  $x$  distinct days, for several different values of  $k$ .

Fig. 3(b) shows the fraction of the total legitimate mail that originates from IP addresses that are in the  $k$ -good set and appear in at least  $x$  days, for each threshold  $k$ .

Most of the IP addresses in a  $k$ -good set are not present very long, and the number of IP addresses falls quickly, especially in the first few days. However, their contribution to the legitimate mail drops much more slowly as  $x$  increases. The result is that the few longer-lived IPs contribute to most of the legitimate mail from a  $k$ -good set. For example, only 5% of all IP addresses in the 20-good set appear at least 10 distinct days, but they contribute to almost 87% of all legitimate mail from a  $k$ -good set. If the  $k$ -good set contributes to a significant fraction of the legitimate mail, then the few longer-lived IP addresses also contribute significantly to the total legitimate mail. For instance, IP addresses in the 20-good set contribute to 63.5% of the total legitimate mail received. Only 2.1% of those IP addresses are present for at least 30 days, but they contribute to over 50% of the total legitimate mail received.

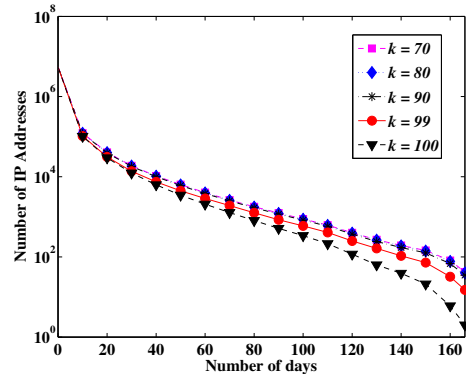
**Result 4. Distribution of legitimate mail from persistent  $k$ -good IPs:** Fig. 3 indicates that (i) IP addresses with low lifetime spam ratios (small  $k$ ) tend to contribute a major proportion of the total legitimate email, and (ii) only a small fraction of the IP addresses with a low lifetime spam-ratio addresses appear over many days, but they contribute to a significant fraction of the legitimate mail.

The graphs also reveal another trend: the longer an IP address lasts, the more stable is its contribution to the legitimate mail. For example, 0.09% of the IP addresses in the 20-good set are present for at least 60 days, but they contribute to over 40% of the total legitimate mail received. From this, we can infer that there were an additional 1.2% of IP addresses in the 20-good set that were present for 30-59 days, but they only contributed to 10% of the total legitimate mail received.

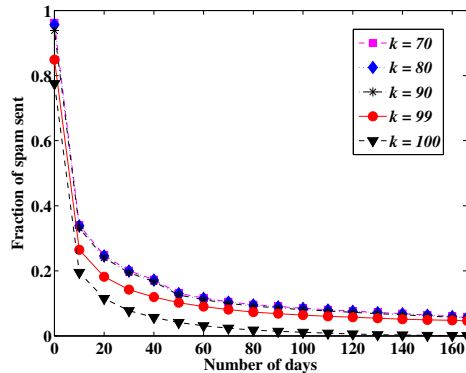
<sup>1</sup>Our analysis considers persistence of IP addresses only in our data set, i.e., it considers whether the IP address has sent mail for  $x$  days to our mail server. These IP addresses may have sent mail to other mail servers on more days, and combining data across multiple different mail servers may give a better picture of stability of IP addresses sending mail. Nevertheless, in this work, we focus on the persistence in one data set, as it highlights behavioural differences due to community structure present within a single vantage point.

Fig. 4 presents a similar analysis of persistence for IP addresses with a high lifetime spam-ratio. Like the  $k$ -good IP addresses and  $k$ -good sets, we define  $k$ -bad IP addresses and  $k$ -bad sets.

**Definition 2.** A  $k$ -bad IP address is an IP address that has a lifetime spam-ratio of at least  $k$ . A  $k$ -bad set is the set of all  $k$ -bad IP addresses.



(a) No. of  $k$ -bad IP addresses present in  $x$  or more days



(b) Fraction of spam sent by  $k$ -bad IP addresses present in  $x$  or more days

Figure 4: Persistence of  $k$ -bad IP addresses.

Fig. 4(a) presents the number of IP addresses in the  $k$ -bad set that are present in at least  $x$  days, and Fig. 4(b) presents the fraction of the total spam sent by IP addresses in the  $k$ -bad set that are present in at least  $x$  days.

**Result 5. Distribution of spam from persistent  $k$ -bad IPs:** Fig. 4 indicates that (i) IP addresses with high lifetime spam ratios (large  $k$ ) tend to contribute almost all of the spam, (ii) most of these high spam-ratio IPs are only present for a short time (this is consistent with the finding in [19]) and account for a large proportion of the overall spam, and (iii) the small fraction of these IPs that do last several days contribute a non-trivial fraction of the overall spam; however, a much larger fraction of spam comes from IP addresses that are not present for

very long. As in the case of the  $k$ -good IP addresses, the spam contribution from the  $k$ -bad IP addresses tends to get more stable with time.

So, for instance, we can see from Fig. 4 that only 1.5% of the IP addresses in the 80-bad set appear in at least 10 distinct days, and these contribute to 35.4% of the volume of spam from the 80-bad set, and 34% of the total spam. The difference is more pronounced for 100-bad IP addresses: 2% of the 100-bad IP addresses appear for 10 or more distinct days, and contribute to 25% of the total spam volume.

The results of this section have implications in designing spam filters, especially for applications where the goal is to prioritize legitimate mail rather than discard spam. While spamming IP addresses that are present sufficiently long can be blacklisted, the scope of a purely blacklisting approach is limited. On the other hand, a very significant fraction of the legitimate mail can be prioritized by using the history of the senders of legitimate mail.

### 2.2.3 Temporal Stability

Next, we seek to understand whether IP addresses in the  $k$ -good set change their daily spam-ratio dramatically over the course of their lifetime. The question we want to answer is: of the IP addresses that appear in a  $k$ -good set (for small values of  $k$ ), what fraction of them have ever had “high” daily spam-ratios, and how often do they have “high” spam-ratios? Thus, we want to understand the *temporal stability* of the spam-ratio of IP addresses in  $k$ -good sets. In this section, we focus on  $k$ -good IP addresses; the results for the  $k$ -bad IP addresses are similar.

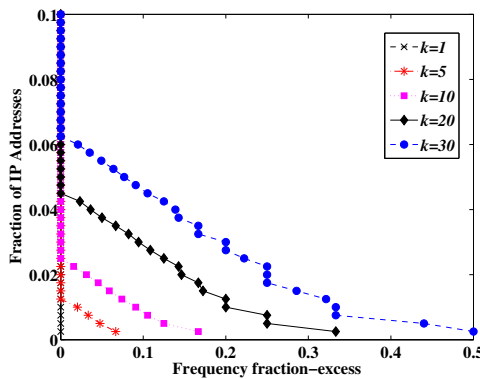


Figure 5: Temporal stability of IP addresses in  $k$ -good sets, shown by CCDF of frequency-fraction excess.

We compute the following metric: for each IP address in a  $k$ -good set, we count how often its daily spam-ratio exceeds  $k$  (and normalize this count by the num-

ber of days it appears). We define this quantity to be the *frequency-fraction excess* of the IP address, for the  $k$ -good set. We plot the complementary cdf (CCDF) of the *frequency-fraction excess* of all IP addresses in the  $k$ -good set.<sup>2</sup> Intuitively, the distribution of the frequency-fraction excess is a measure of how many IP addresses in the  $k$ -good set exceed  $k$ , and how often they do so.

Fig. 5 shows the CCDF of the frequency-fraction excess for several  $k$ -good sets. It shows that the majority of the IP addresses in each  $k$ -good set have a frequency-fraction excess of 0, and that 95% of the  $k$ -good IP addresses have a frequency-fraction excess of at most 0.1.

We explain the implications of Fig. 5 to the temporal stability of the spam-ratio of IP addresses with an example. We focus on the  $k$ -good set for  $k = 20$ : this is the set of IP addresses whose lifetime spam-ratio is bounded by 20%. We note that the frequency-fraction excess is 0 for 95% of the 20-good IP addresses. This implies that 95% of IP addresses in this  $k$ -good set do not send more than 20% spam *any* day, i.e., every time they appear, they have a daily spam-ratio of at most 20%. We also note that fewer than 1% of the IP addresses in this  $k$ -good set have a frequency-fraction excess larger than 0.2.

Thus, for many  $k$ -good sets with small  $k$ -values, only a few IP addresses have a significant frequency-fraction excess, i.e., very few IP addresses in those sets exceed the value  $k$  often. Since they would need to exceed  $k$  often to change their spamming behaviour significantly, it follows that most IP addresses in the  $k$ -good set do not change their spamming behaviour significantly.

In addition, the frequency-fraction excess is perhaps too strict a measure, since it is affected even if  $k$  is exceeded slightly. We also compute a similar measure that increases only when  $k$  is exceeded by 5%. No more than 0.01% of IP address in the  $k$ -good set exceed  $k$  by 5%, for any  $k \leq 30\%$ . Since we are especially interested in the temporal stability of IP addresses that appear often, we compute also the frequency-fraction excess distribution for IP addresses that appear for 10, 20, 40 and 60 days. In each case, almost no IP address exceeds  $k$  by more than 5%, for any  $k \leq 30\%$ .

We summarize this discussion in the following result.

**Result 6. Temporal stability of  $k$ -good IPs:** *Fig. 5 shows that most IP addresses in  $k$ -good sets (for low  $k$ , e.g.,  $k \leq 30\%$ ) do not exceed  $k$  often; i.e., most  $k$ -good IP addresses have low spam-ratios (at most  $k$ ) nearly every day.*

With the above result, we can analyze the behaviour of  $k$ -good sets of IP addresses, constructed over their entire lifetime, and their behaviour in shorter time intervals.

<sup>2</sup>That is, we plot the fraction of IP addresses in the  $k$ -good set whose frequency-fraction excess is at least  $x$ . The  $y$ -axis of the plot is restricted for readability.

The analysis of these three properties of IP addresses indicates that a significant fraction of the legitimate mail comes from IP addresses that persistently appear in the traffic. These IP addresses tend to exhibit stable behaviour: they do not fluctuate significantly between sending spam and legitimate mail. These results lend weight to our hypothesis that spam mitigation efforts can benefit by preferentially allocating resources to the stable and persistent senders of legitimate mail. However, there is still a substantial portion of the mail that cannot be accounted for through only IP address-based analysis. In the next section, we focus on how to account for this mail.

### 3 Analysis of Cluster Characteristics

So far, we have analyzed whether the historical behaviour of individual IP addresses can be used to distinguish between senders of legitimate mail and spammers. However, if we only consider the history of individual IP addresses, we cannot determine whether a new, previously unseen, IP address is likely to be a spammer or a sender of legitimate mail. If there are many such IP addresses, then, in order to be useful, any prioritization scheme would need to assign these new IP addresses appropriate reputations as well. Indeed, in Sec. 2.2.2, we found that most IP addresses sending mail are short-lived and that such short-lived IPs account for a significant proportion of both legitimate mail and spam. Any prioritization scheme would thus need to be able to find reputations for these IP addresses as well.

To address this issue, we explore whether coarser aggregations of IP addresses exhibit more persistence and afford more effective discriminatory power for spam mitigation. If such aggregations of IP addresses can be found, the reputation of an unseen IP address could be *derived* from the historical reputation of the aggregation they belong to.

We focus on IP aggregations given by *network-aware clusters* of IP addresses [15]. Network-aware clusters are sets of unique network IP prefixes collected from a wide set of BGP routing table snapshots. In this paper, an IP address belongs to a network-aware cluster if the longest prefix match of the IP address matches the prefix associated with the cluster. In the reputation mechanisms we explore in Sec. 4, an IP address derives the reputation of the network-aware cluster that it belongs to. We use network-aware clustering because these clusters represent IP addresses that are close in terms of network topology and do, with high probability, represent regions of the IP space that are under the same administrative control and share similar security and spam policies [15].

In this section, we present measurements suggesting that network-aware clusters of IP addresses may provide

a good basis for reputation-based classification of IP addresses. We focus on the following questions:

- *Granularity*: Does the mail originating from network-aware clusters consist of mostly spam or mostly legitimate mail, so that these clusters could be useful as a reputation-granting mechanism for IP addresses?
- *Persistence*: Do individual network-aware clusters appear (i.e., do IP addresses belonging to the clusters appear) over long periods of time, so that network-aware clusters could potentially afford us a useful mechanism to distinguish between different kinds of ephemeral IP addresses?

As in the IP-address case, we adopt the spam-ratio of a network-aware cluster as the discriminating feature of clusters and examine whether clusters with low/high spam-ratios are granular and persistent.

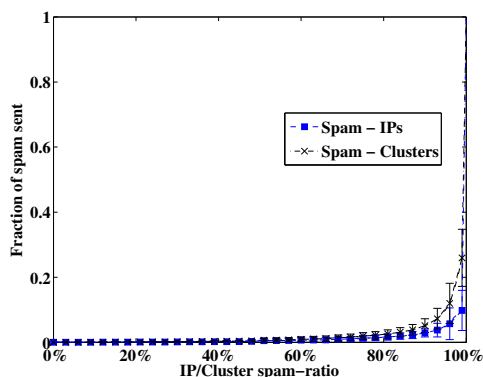
Before examining these two properties in detail, we first summarize our analysis of the properties with respect to which clusters behave as IP addresses do: *clusters turn out to be at least as (and usually more) temporally stable as IP addresses* (similar to the IP address behaviour explored in Sec. 2.2.3), which is the expected behaviour; *the distribution of clusters by daily cluster spam-ratio is similar to the distribution of IP addresses by IP spam-ratio* (similar to the IP address behaviour explored in Sec. 2.2.1).

#### 3.1 Cluster Granularity

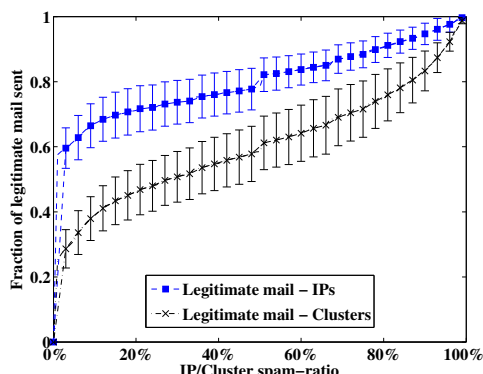
For network-aware clustering of IP addresses to be useful, the clusters need to be sufficiently homogeneous in terms of their legitimate mail/spam behavior so that the cluster information can be used to separate the bulk of legitimate mail from the bulk of spam. Recall that with the IP addresses, we analyzed the extent to which IP spam-ratios could be used to identify the IP addresses sending the bulk of legitimate mail and spam. Here, we analyze whether, instead of an IP's individual spam-ratio, the spam-ratio of the parent cluster can be used for the same purpose.

To do so, we need to understand how well the cluster spam-ratio approximates the IP spam-ratio. In our context, we focus on the following question: can we still distinguish between the IP addresses that send the bulk of the legitimate mail and the bulk of the spam? If we can, within a margin of error, it would suggest that cluster-level analysis is nearly as good as IP-level analysis.

For the analysis here, we determine the spam-ratio of each cluster by analyzing the mail sent by all IP addresses belonging to that cluster and assign to IP addresses the spam-ratios of their respective clusters. In



(a) Fraction of spam sent by clusters & IPs, as a function of cluster & IP spam-ratios.



(b) Legitimate mail sent by clusters & IPs, as a function of cluster & IP spam-ratios.

Figure 6: Penalty of using cluster-level analysis.

the rest of this discussion, we will refer to legitimate mail/spam sent by IP addresses belonging to a cluster as the legitimate mail/spam *sent by* or *coming from* that cluster. As with the IP-based analysis, we examine how the volume of legitimate mail and spam from IP addresses is distributed as a function of their cluster spam-ratios. To understand the additional error imposed by using the cluster spam-ratio, we compare it with how those volumes are distributed as a function of the IP spam-ratio.

Fig. 6(a) shows how the spam sent by IP addresses with a cluster or IP spam-ratio of at most  $k$  varies with  $k$ . Specifically, on day  $i$ , let  $CS_i(k)$  and  $IS_i(k)$  be the fraction of spam sent by the IP addresses with a cluster spam-ratio (and IP spam-ratio, respectively) of at most  $k$ . Fig. 6(a) plots  $CS_i(k)$  and  $IS_i(k)$  averaged over all the days in the data set, as a function of  $k$ , along with confidence intervals.

**Result 7. Distribution of spam with cluster and IP spam-ratios:** Fig. 6(a) shows that almost all (over 95%) of the spam every day comes from IPs in clusters with a very high cluster spam-ratio (over 90%). A similar frac-

tion (over 99% on average) of the spam every day comes from IP addresses with a very high IP spam-ratio (over 90%).

This suggests that spammers responsible for a high volume of the total spam may be closely correlated with the clusters that have a very high spam-ratio. The graph indicates that if we use a spam-ratio threshold of  $k \leq 90\%$  for spam mitigation, then using the IP spam-ratio rather than the corresponding cluster spam-ratio as the discriminating feature would increase the amount of spam identified by less than 2%. This suggests that cluster spam-ratios are a good approximation to IP spam-ratios for identifying the bulk of the spam sent.

We next consider how legitimate mail is distributed with the cluster spam-ratios and compare it with IP spam-ratios (Fig. 6(b)). We compute the following metric: Let  $CL_i(k)$  and  $IL_i(k)$  be the fraction of legitimate mail sent by IPs with cluster and IP spam-ratios of at most  $k$  on day  $i$ . Fig. 6(b) plots  $CL_i(k)$  and  $IL_i(k)$  averaged over all the days in the data set as a function of  $k$ , along with confidence intervals.

**Result 8. Distribution of legitimate mail with cluster and IP spam-ratios:** Fig. 6(b) shows that a significant amount of legitimate mail is contributed by clusters with both low and high spam-ratios. A significant fraction of the legitimate mail (around 45% on average) comes from IP addresses with a low cluster spam-ratio ( $k \leq 20\%$ ). However, a much larger fraction of the legitimate mail (around 70%, on average) originates from IP addresses with a similarly low IP spam-ratio.

The picture here, therefore, is much less promising: even when we consider spam-ratios as high as 30 – 40%, the cluster spam-ratios can only distinguish, on average, around 50% of the legitimate mail. By contrast, IP spam-ratios can distinguish as much as 70%. This suggests that IP addresses responsible for the bulk of legitimate mail are much less correlated with clusters of low spam-ratio.

We can then make the following conclusion: suppose we use a classification function to accept or reject IP addresses based on their cluster spam-ratio. What additional penalty would we incur over a similar classification function that used the IP address’s own spam-ratio? Fig. 6(b) suggests that, if the threshold is set to 90% or higher, we incur very little penalty in both legitimate mail acceptance and spam. However, if the threshold is set to 30 – 40%, we may incur as much as a 20% penalty in doing so.

However, there are two additional ways in which such a classification function could be enhanced. First, as we have seen, the bulk of the legitimate mail does come from persistent  $k$ -good IP addresses. This suggests that we could potentially identify more legitimate mail by considering the persistent  $k$ -good IP addresses in addition

to cluster-level information. Second, for some applications, the correlation between high cluster spam-ratios and the bulk of the spam may be sufficient to justify using cluster-level analysis. For example, under the existing distribution of spam and legitimate mail, even a high cluster spam-ratio threshold would be sufficient to reduce the total volume of the mail accepted by the mail server. This is exactly the situation in the server overload problem and we see the effect in the simulations in Sec. 4.

### 3.2 Persistence

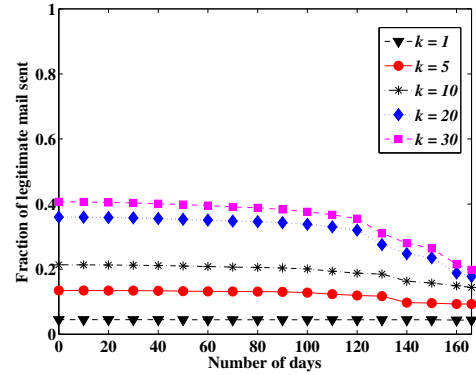
Next, we explore how persistent the network-aware clusters are, just as we did for the IP addresses. We define a cluster to be *present* on a day if at least one IP address that belongs to that cluster appears that day. We reported earlier that we found the clusters themselves to be at least as (and usually more) temporally stable as IP addresses. Our next goal is to examine how much of the total legitimate mail/spam the long-lived clusters contribute.

As in Sec. 2.2.2, we will define  $k$ -good and  $k$ -bad clusters; to do that, we use the *lifetime cluster spam-ratio*: the ratio of the total spam sent by the cluster to the total mail sent by it over its lifetime.

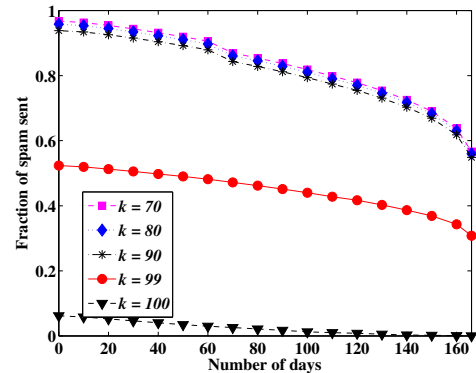
**Definition 3.** A  $k$ -good cluster is a cluster of IP addresses whose lifetime cluster spam-ratio is at most  $k$ . The  $k$ -good cluster-set is the set of all  $k$ -good clusters. A  $k$ -bad cluster is a cluster of IP addresses whose lifetime cluster spam-ratio is at least  $k$ . The  $k$ -bad cluster-set is the set of all  $k$ -bad clusters.

Fig. 7(a) examines the legitimate mail sent by  $k$ -good clusters for small values of  $k$ . We first note that the  $k$ -good clusters (even when  $k$  is as large as 30%) contribute less than 40% of the total legitimate mail; this is in contrast to, for instance, 20-good IP addresses that contributed to 63.5% of the total legitimate mail. However, we note the contribution from long-lived clusters is far more than from long-lived individual IPs. The difference from Fig. 3(b) is striking: e.g.,  $k$ -good clusters present for 60 or more days contribute to nearly 99% of the legitimate mail from the  $k$ -good cluster set. So, any cluster accounting for a non-trivial volume of legitimate mail is present for at least 60 days. Indeed, the legitimate mail sent by  $k$ -good clusters drops to 90% of  $k$ -good cluster-set's total only when restricted to clusters present for 120 or more days; by contrast, for individual IP addresses, the legitimate mail contribution dropped to 87% of the 20-good set's total after just 10 days.

Fig. 7(b) presents the same analysis for  $k$ -bad clusters. Again, there are noticeable differences from the  $k$ -bad IP addresses, and also from the  $k$ -good clusters. A much larger fraction of spam comes from long-lived clusters than from long-lived IPs in Fig. 4(b). For example, over



(a) Fraction of legitimate mail sent by  $k$ -good clusters that appear in at least  $x$  days



(b) Fraction of spam sent by  $k$ -bad clusters that appear in at least  $x$  days

Figure 7: Persistence of network-aware clusters.

92% of the total spam is contributed by 90-bad clusters present for at least 20 days. This is in sharp contrast with the  $k$ -bad IP addresses, where only 20% of the total spam comes from IP addresses that last 20 or more days. We also note that the 90-bad cluster-set contributes to nearly 95% of the total spam. Thus, in contrast to the legitimate mail sent by  $k$ -good cluster-sets, the bulk of the spam comes from the  $k$ -bad cluster-sets with high  $k$ .

**Result 9. Distribution of mail from persistent clusters:** Fig. 7 shows that the clusters that are present for long periods with high cluster spam-ratios contribute the overwhelming fraction of the spam sent, while those present for long periods with low cluster spam-ratios contribute a smaller, though still significant, fraction of the legitimate mail sent.

The above result suggests that network-aware clustering can be used to address the problem of transience of IP addresses in developing history-based reputations of IP addresses: even if individual IP addresses are ephemeral, their (possibly collective) history would be useful in assigning reputations to other IP addresses originating

from the same cluster.

## 4 Spam Mitigation under Mail Server Overload

In the previous section, we have demonstrated that there are significant differences in the historical behaviour of IP addresses that send a lot of spam, and those that send very little. In this section, we consider how these differences in behaviour could be exploited for spam mitigation.

Our measurements have shown that senders of legitimate mail demonstrate significant stability and persistence, while spammers do not. However, the bulk of the high volume spammers appear to be clustered well within many persistent network-aware clusters. Together, these suggest that we can design techniques based on the historical reputation of an IP address and the cluster to which it belongs. However, because mail rejection mechanisms necessarily need to be conservative, we believe that such a reputation-based mechanism is primarily useful for prioritizing legitimate mail, rather than actively discarding all suspected spammers.

As an application of these measurements, we now consider the mail-server overload problem described in the introduction. In this section, we demonstrate how the problem could be tackled with a reputation-based mechanism that exploits these differences in behaviour. In Sec. 4.1, we explain the mail-server overload problem in more detail. In Sec. 4.2, we explain our approach, describing the mail server simulation and algorithms that we use, and in Sec. 4.3, we present an evaluation showing the performance improvement gained using these differences in behaviour.

We emphasize that this simulation study is intended to demonstrate the potential of using these behavioural differences in the legitimate mail and spam for prioritizing exclusively by IP addresses. However, it is *not* intended to be comparable to content-based spam filtering. We also note that these differences in behaviour could be applied in other ways as well and at other points in the mail processing as well. The quantitative benefits that we achieve may be specific to our application and may be different in other applications.

### 4.1 Server Overload Problem

The problem we consider is the following: When the mail server receives more SMTP connections than it can process in a time interval, how can it selectively accept connections to maximize the acceptance of legitimate mail? That is, the mail server receives a sequence of connection requests from IP addresses every second, and each connection will send mail that is either legitimate or

spam. Whether the IP address sends spam or legitimate mail in that connection is not known at the time of the request, but is known after mail is processed by the spam filter. The mail server has a finite capacity of the number of mails that can be processed in each time interval, and may choose the connections it accepts or rejects. The goal of the mail server is to selectively accept connections in order to maximize the legitimate mail accepted.

We note that spammers have strong incentive to cause mail servers to overload, and illustrate this with an example. Assume that a mail server can process 100 emails per second, that it will start dropping new incoming SMTP connections when its load reaches 100 emails per second, and that it crashes if the offered load reaches 200 emails per second. Assume also that 20 legitimate emails are received per second. A spammer could increase the load of the mail server to 100% by sending 80 emails per second which would be all received by the mail server. Alternatively, the spammer could also increase the load to 199%, by sending 179 spam emails per second, and now nearly half the requests would not be served. If the mail server is unable to distinguish between the spam requests and the legitimate mail requests, it drops connections at random, and the spammer will be able to successfully get through 89 spam emails per second to the mail server, as compared to the 80 in the previous case.

Thus, the optimal operation point of a spammer, assuming that he has a large potential sending capacity, is not the maximum capacity of the mail server but the maximum load before the mail server will crash. This observation indicates that the approach of throwing more resources at the problem would only work if the mail server capacity is increased to exceed the largest botnet available to the spammer. This is typically not economically feasible and a different approach is needed.

The results in Sec. 2 and Sec. 3 suggest that there may be a history-based reputation function  $R$ , that relates IP addresses to their likelihood of sending spam. Thus, for example, if  $R(i)$  is the probability that an IP address  $i$  sends legitimate mail, then maximizing the quantity  $\sum R(i)$  would maximize the expected number of accepted legitimate mail. If the reputation function  $R$  were known, this problem would be similar to admission control and deadline scheduling; however, in our case,  $R$  is not known.

In this work, we choose one *simple* history-based reputation function and demonstrate that it performs well. We reiterate that our goal is *not* to explore the space of the reputation functions or to find the best reputation function. Rather, our goal is to demonstrate that they could potentially be used to increase the legitimate mail accepted when the mail-server is overloaded. In addition, our goal is to preferentially accept e-mails from certain IP addresses *only* when the mail servers are overloaded

– we would like to minimize the impact on mail servers when they are not overloaded. A poor choice of  $R$  will then not impact the mail server under normal operation.

The techniques and the reputation functions that we choose address concerns that are different from those addressed by standard IP-based classification techniques like blacklisting and greylisting, as neither blacklisting nor greylisting would directly solve the server overload problem. Blacklisting has well-known issues: building a blacklist takes time and effort, most IP addresses that send spam are observed to be ephemeral, appearing very few times, and many of them are not even present in any single blacklist.

While greylisting is an attractive short-term solution that has been observed to work quite well in practice, it is not robust to spammer evasion, since spammers could simply mimic the behaviour of a normal mail server. Greylisting aims to optimize a different goal – its goal is to delay the mail in the hope that a spam signature is generated in the mean time, so that spam can be distinguished from non-spam; however, delaying the mail does not reduce the overall server load, since the spammer can always return to send more mail, and computing a content-based spam signature would continue to be as expensive. Indeed, greylisting gives spammers even more incentive to overload mail servers by re-trying after a specified time period.

Our techniques for the server overload problem provide an additional layer of information when compared to blacklisting and greylisting. It may be possible to use the IP structure information to enhance greylisting, to decide, at finer granularities and with soft thresholding, which IP addresses to deny.

## 4.2 Design and Algorithms

Today, when mail servers experience overload, they drop connections greedily: the server accepts all connections until it is at maximum load, and then refuses all connection requests until its load drops below the maximum. We aim to improve the performance under overload by using information in the structure of IP addresses, as suggested by the results in Sec. 2 and Sec. 3. At a high-level, our approach is to obtain a history of IP addresses and IP clusters, and use it to select the IP addresses that we prioritize under overload. To explore the potential benefits of this approach, we simulate the mail server operation and allow some additional functionality to handle overload.

To motivate our simulation, we describe briefly the way many mail servers in corporations and ISPs operate. First, the sender's mail server or a mail relay tries to connect to the receiving mail server via TCP. The receiving mail server accepts the connection if capacity is avail-

able, and then the mail servers perform the SMTP handshake and transfer the email. The receiving mail server stores the email to disk and adds it to the spam processing queue. For each e-mail on the queue, the receiving mail server then performs content-based spam filtering [3, 1] which is typically the most expensive part of email processing. After this, the spam emails are dropped or delivered to a spam mailbox, and the good emails are delivered to the inbox of the recipient.

In our simulation we simplify the mail server model, while ensuring that it is still sufficiently rich to capture the problem that we explore. We believe that our model is sufficiently representative for a majority of mail server implementations used today; however, we acknowledge that there are mail server architectures in use which are not fully captured in our model. In the next section, we describe the simulation model in more detail.

### 4.2.1 Mail Server Simulation

We simulate mail-server operation in the following manner:

- *Phase 1:* When the mail server receives an SMTP connection request, it may decide whether or not to accept the connection. If it decides to accept the connection, the incoming mail takes  $t$  time units to be transferred to the mail server. Thus, if a server can accept  $k$  connection requests simultaneously, it behaves like a  $k$ -parallel processor in this phase. We do so because this phase models the SMTP handshake and transfer of mail, and therefore, it needs to model state for each connection separately.
- *Phase 2:* Once the mail has been received, it is added to a queue for spam filtering and delivery to the receiving mailbox if any. At each time-step, the mail server selects mails from this queue and processes them; the number of mails chosen depend on the mail server's capacity and the cost of each individual mail. Here, since we model computation cycles, a sequential processing model suffices. The mail server has a timeout: it discards any mail that has been in the queue for more than  $m$  time units. If the load has sufficient fluctuation, a large timeout would be useful, but we want to minimize timeout since email has the expectation of being timely.

We assume that the cost of denying/dropping a request is 0, the cost of processing the SMTP connection is  $\alpha$  fraction of its total cost, and the cost of the remainder is  $1 - \alpha$  fraction of the total cost. We also allow Phase 1 of the mail server simulator to have  $\alpha$  fraction of the server's computational resources, and Phase 2 to have the remainder. Since the content-based analysis is typ-

ically the most expensive part of processing a message, we expect that  $\alpha$  is likely to be small.

This two-phase simulation model allows for more flexibility in our policy design, since it opens the possibility of dropping emails which have already been received and are awaiting spam filtering without wasting too many resources.

#### 4.2.2 Policies

Next, we present the prioritization/drop policies that we implemented and evaluated on the mail server simulator. In this simulation model, the default mail-server action corresponds to the following: at each time-interval, the server accepts incoming requests in the order of arrival, as long as it is not overloaded. Once mail has been received, the server processes the first mail in the queue, and discards any mail that has exceeded its timeout. We refer to this as the *greedy* policy.<sup>3</sup>

The space of policy options that a mail-server is allowed to operate determine the kinds of benefits it can get. In this problem, one natural option for the mail server is to decide immediately whether to accept or reject a connection request. However, such a policy may be quite sensitive to fluctuation in the workload received at the mail server. Another option may be to reject some e-mails *after* the SMTP connection has been accepted, but *before* any spam-filtering checks or content-based analysis (such as spam-filtering software) has been applied. Note that content-based analysis typically is the most computationally expensive part of receiving mail. Thus, with this option, the mail server may do a small amount of work for some additional emails that eventually get rejected, but is less affected by the fluctuation of mail arrival workload. We restrict the space of policy options to the time before *any* content-based analysis of the incoming mail is done.

To solve the mail-server overload problem, we implement the following policies at the two phases:

- *Phase-1 policy*: The policy in Phase 1 is designed to preferentially accept IP addresses with a good reputation when the server is near maximum load: as the server gets closer to overload, the policy only accepts IP addresses with better and better reputations. The policy itself is more complex, since it needs to consider the expected legitimate mail workload, and yet not stay idle too long. We therefore leave exact details to the appendix. In addition, when the load is below some percentage (we choose 75%) of the

<sup>3</sup>To ensure that the current mail server policy is not unfairly modelled under this simulation model, we evaluated greedy policies in another simulation model, in which each connection took  $z$  time units to process from start to end. The performance of the greedy policy was similar, therefore we do not describe the model further.

total capacity, the server accepts all mail: this way, it minimizes impact on normal operation of the mail server.<sup>4</sup>

- *Phase-2 policy*: The scheduling policy here is easier to design, since the queue has some knowledge of what needs to be processed. Even a simple policy that greedily accepts the item with the highest reputation value will do well, as long as the reputation function is reasonably accurate. We use this greedy policy for Phase 2.

Our history-based reputation function  $R$  is simple: First, we find a list of persistent senders of legitimate mail from the same time period (we choose all senders that have appeared in at least 10 days), and for these IP addresses, we use their lifetime IP spam-ratio as their reputation value. For the remaining IP addresses, we use their cluster spam-ratio as their reputation value: for each week, we use the history of the preceding four weeks in computing the lifetime spam-ratio (defined over 4 weeks) for each cluster that sends mail.<sup>5</sup> In this way, we combine the results of the IP-based analysis and cluster-based analysis in Sec. 2 in designing the reputation function.

This reputation function is extremely simple, but it still illustrates the value of using a history-based reputation mechanism to tackle the mail server overload problem. We also note that the historical IP reputations based on network-aware clusters in this manner may not always be perfect predictors of spamming behaviour. While network-aware clusters are an aggregation technique with a basis in network structure, they could serve as a starting point for more complex clustering techniques, and these techniques may also incorporate finer notions of granularity and confidence.

A more sophisticated approach to using the history of IP addresses and network-aware clusters that addresses these concerns is likely to yield an improvement in performance, but is beyond the scope of this paper and left as future work. In the following section, we describe the performance benefits that we gain from using this reputation function in the evaluation.

### 4.3 Evaluation

We evaluate our history-based policies by replaying the traces of our data set on our simulator. Since the traces record each connection request with a time-stamp, we can replay the traces to simulate the exact workload received by the mail server. We do so, with the simplifying

<sup>4</sup>Technically, this is slightly more complex: it examines if the load is below 75% of the server capacity allowed to Phase 1.

<sup>5</sup>One technical detail left to consider are the IP addresses originating from clusters without history. In our reputation function, any IP address that has no history-based reputation value is given a slightly bad reputation.

assumption that each incoming e-mail incurs the same computational cost. Since our traces are fixed, we simulate overload by decreasing the simulated server's capacity, and replaying the same traces. This way, we do not change the distribution and connection request times of IP addresses in the input traces between the different experiments. At the same time, it allows us to simulate, without changing the traces, how the mail server behaves as a function of the increasing workload.

*Simulation Parameters:* We now explain the parameters that we choose for our simulation. We choose the time  $t$  for the Phase 1 operation to be 4s.<sup>6</sup> We use 60s for the timeout  $m$ , the waiting time in the queue before Phase 2 (it implies that mail will be delivered within 1 minute, or discarded after Phase 1). This appears to be sufficiently small so as to not noticeably affect the delivery of legitimate mail.<sup>7</sup>

To induce overload, we vary the capacity of the simulated mail server to 200, 100, 66, 50, and 40 messages/minute. The greedy policy processed an average of 95.2% of the messages received when the server capacity was set to 200 messages/minute, as seen in Table 2. At capacities larger than 200 messages/minute, the number of messages processed by the greedy policy grows very slowly, indicating that this is likely to be an effect of the distribution of connection requests in the traces. For this reason, we take capacity of 200/minute as the required server capacity. We then refer to the other server capacities in relation to required server capacity for this trace workload: a server with capacity of 100 messages/minute must process the same workload with half the capacity of the required server, so we define it to have an *overload-factor* of 2. Likewise, the server capacities we test 200, 100, 66, 50 and 40 messages/minute have overload-factors of around 1, 2, 3, 4, and 5 respectively.

Recall that the parameter  $\alpha$  is the cost of processing the message at Phase 1. We expect  $\alpha$  to impact the performance, so we test two values  $\alpha = 0.1, 0.5$  in the evaluation; recall that  $\alpha$  is likely to be small, and so  $\alpha = 0.5$  is a conservative choice here. The value of  $\alpha$  has no effect on the performance of the greedy policy. For this reason, the discussion features only one greedy policy for all values of  $\alpha$ . For the history-based policies,  $\alpha$  sometimes has an effect on the performance, since these policies allow for a decision to be taken at Phase 2. We therefore refer to the history-based policies as 10-policy,

<sup>6</sup>We vary  $t$  for Phase 1 between 2-4s: our traces have a recorded time granularity of 1s, and the maximum seen in the traces before a disconnect was 4s. This does not appear to impact the results presented here, since both kinds of policies receive the same value of  $t$ . We present in the results for  $t = 4s$

<sup>7</sup>This value also has no noticeable impact on our results when  $m \geq 20s$  suggesting that most of the legitimate mail is processed quickly, or not at all.

and 50-policy, for  $\alpha = 0.1$  and 0.5 respectively.

#### 4.3.1 Impact on Legitimate mail

We first compare the number of legitimate mails accepted by the different policies over many time intervals, where each interval is an hour long. Since our goal is to maximize the amount of legitimate mail accepted, the primary metric we use is the *goodput ratio*: the ratio of legitimate mail accepted by the mail server to the total legitimate mail in the time interval. This is a natural metric to use, since it makes the different time intervals comparable, and so we can see if the policies are consistently better than the greedy policy, rather than being heavily weighted by the number of legitimate mails in a few time intervals. For the performance evaluation, we examine the average goodput ratio, the distribution of the goodput ratios and the goodput improvement factor.

*Average Goodput Ratio:* Table 1 shows the average goodput ratios for the different policies under different levels of overload. It shows that, on average, for each of these overloads, the goodput of any of the policies is better than the Greedy policy. The difference is marginal at overload-factor 1, and increases quickly as the overload-factor increases: at overload-factor 4, the average goodput ratio is 64.3–64.5% for any of the history-based policies, in comparison to 26.8% for the greedy policy. We also observe that the history-based policies scale more gracefully with the overload. Thus, we conclude that, on average, the history-based policies gain a significant improvement over the greedy policy.

*Distribution of Goodput Ratios:* While the average goodput ratio is a useful summarization tool, it does not give a complete picture of the performance. For this reason, we next compare the *distribution* of the server goodput in the different time intervals. Fig. 8(a)-(b) shows the CDF of the goodput ratios for the different policies, for two overload-factors: 1 and 4. We observe that the goodput ratio distributions are quite similar for the greedy and history-based policies when the overload-factor is 1 (Fig. 8(a)): over 40% of the time, all of the policies accept 100% messages. This changes drastically as the overload-factor increases. Fig. 8(b) shows the goodput ratio distributions for overload-factor 4. As much as 50% of the time, the greedy policy has a goodput-ratio of at most 0.2. By contrast, more than 90% of the time, the history-based policies have a goodput ratio of at least 0.5. The results show that the the history-based policies have a consistent and significant improvement over the greedy policy when the load is sufficiently high.

*Improvement factor of Goodput-Ratios:* Finally, we compare the goodput ratios on a per-interval basis. For this analysis, we focus on the 10-policy; our goal is to

see *how often* the 10-policy does better than the greedy algorithm. That is, for each time interval, we compute the *goodput-factor*, defined to be  $\frac{\text{Goodput of 10-Policy}}{\text{Goodput of Greedy}}$ . Fig. 8(c) plots how often goodput-factor lies between 90% – 300% for the different overload-factors. We note that when the overload-factor is 1, the performance impact of our history-based policy on the legitimate mail is marginal: in all the time intervals, the 10-policy has a goodput-factor of 90%, and 99% of the time, it has a goodput factor of 99%. As the overload-factor increases, the amount of time intervals in which the 10-policy has a goodput-factor of 100% or more increases, meaning the number of time intervals in which the 10-policy does better than the Greedy algorithm increases, as we would expect. When the overload-factor is 4, for example, 66% of the time, the goodput-factor is 200%: 10-policy accepts at least twice as many legitimate mail. We conclude that in most time intervals, the history-based policies perform better than the greedy policy, and the factor of their improvement increases as the overload-factor increases.

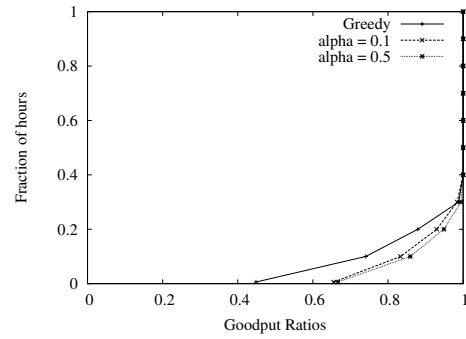
Lastly, we note that the behaviour of the 10-policy and the 50-policy does not appear to differ too much when the overload-factor is sufficiently high or sufficiently low. With intermediate overload-factors, they perform slightly differently, as we see in Table 1: the 50-policy tends to be a little more conservative about accepting messages that may not have a good reputation in comparison to the 10-policy.

#### 4.3.2 Impact on Throughput and Spam

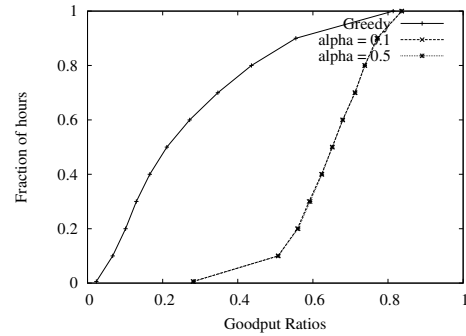
While our primary metric of performance is the goodput, we are still interested in the impact of using the history-based policies on the total messages and spam processed by the mail server. While these are not our primary goals, they are still important since they give a picture of the complete effect of using these history-based policies.

*Impact on Server Throughput:* The history-based policies obviously gain their improvement by selectively choosing the IP addresses to process: it selectively accepts only good IP addresses in the incoming workload, if it is likely that the whole workload might not be processed. This may result in a decrease in server throughput in comparison to the greedy policy for certain load. For example, if the server receives a little less workload than it could process, the history-based policies may process fewer messages than the greedy policy, because they may reserve capacity for good IP addresses that they expect to see but which never actually appear. We observe this in our simulations and we discuss it now.

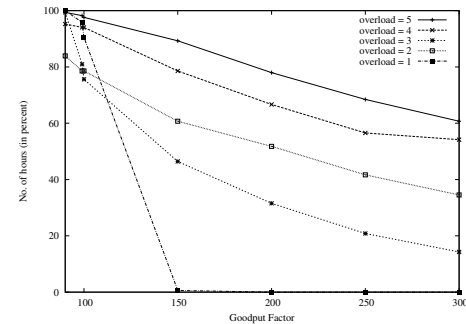
We define *throughput* to be fraction of the total messages processed by the server. Table 2 shows the average throughput achieved by both policies under various capacities of the server. At overload-factor 1, when



(a) Overload-Factor 1: CDF of goodput-ratios for all policies



(b) Overload-Factor 4: CDF of goodput-ratios for all policies



(c) Goodput factor

Figure 8: (a) and (b): CDF of the goodput-ratios for two different overload-factors. (c) shows performance improvement (goodput-factor) for the 10-policy for various overload factors

Overload Factor	Greedy	$\alpha = 0.1$	$\alpha = 0.5$
5	20.3	63	63.6
4	26.8	64.3	64.5
3	39.5	70.7	68.6
2	61.7	84.4	79.6
1	93.7	96	96.7

Table 1: Server Goodput (average, in %).

Overload Factor	Greedy	$\alpha = 0.1$	$\alpha = 0.5$
5	31.6	16.6	16.8
4	39.1	17	17
3	51.6	24.1	22.1
2	71.4	65.8	51.3
1	95.2	93.9	95

Table 2: Server throughput (average, in %).

Overload Factor	Greedy	$\alpha = 0.1$	$\alpha = 0.5$
5	32	14.8	14.9
4	39.5	15.1	15.1
3	52	20.1	15.2
2	71.7	65.2	50.2
1	95.2	93.8	94.9

Table 3: Spam accepted (average, in %).

the greedy algorithm achieves an average throughput of 95%, the history-based policy algorithm achieves an average throughput of 93%. However, even at this point, the history-based policies accept a little more legitimate mail (on average) than the greedy policy. Note that by design, the history-based policies guarantee that when the server receives no more than 75% of its maximum load capacity, its performance is no different from normal.

*Impact on Spam:* We also explored the effect of the history-based policies on the number of spam messages accepted. Table 3 shows the average fraction of spam messages accepted by the policies under various overload factors. We see with an overload-factor of 1, the history-based policies accept only 0.3 – 1% less spam than the Greedy algorithm. As the overload-factor increases and the history-based policies grow more and more conservative in accepting suspected spam, the amount of spam accepted will decrease. For example, at a overload-factor of 2, this drops to 50.2% – 65.5% for the history-based policies. When the overload-factor increases to 4, the history-based policies accept less than 1/2 of the amount of spam accepted by the greedy policy. This suggests that if the server receives much more workload than it can process, the spam is affected much more than the legitimate mail. Therefore, the spammer would not have an incentive to increase the workload significantly, since it is the spam that gets most affected.

Thus, we have shown that our history-based policies achieve a significant and consistent performance improvement over the greedy policy when the server is under overload: we have seen this with multiple metrics of the goodput ratio. We have also seen that the history-based policies do not impact the performance of the server too much when the server is *not* under over-

load. Finally, we have seen that the the spam is indeed affected when the server is significantly overloaded; this is precisely the behaviour we want to induce.

## 5 Related Work

Since spam is so pervasive, much effort has been expended in developing techniques that mitigate spam, and studies that understand various characteristics of spammers. In this section, we briefly survey some of the most related work. We first describe spam mitigation approaches and how they may relate to our work on the server overload problem. Then we discuss measurement studies that are related and complementary to our measurement work.

Traditionally, the two primary approaches to spam mitigation have used content-based spam-filtering and DNS blacklists. Content-based spam-filtering software [3, 1] is typically applied at the end of the mail processing queue, and there has been a lot of research [20, 17, 7, 16] in techniques for content-based analysis and understanding its limits. Agarwal et al. [6] propose content-based analysis to rate-limit spam at the router; this also reduces the load on the mail server, but is not useful for our situation as it may be too computationally expensive.

*DNS blacklists* [4, 5] and are another popular way to reduce spam. Studies on DNS blacklists [14] have shown that over 90% of the spamming IP addresses were present in at least one blacklist at their time of appearance. Our approach is complementary to traditional blacklisting, and the more recent greylisting [13] techniques – we aim to prioritize the legitimate mail, and use the history of IP addresses to identify potential spammers.

Perhaps the closest in spirit to our work in mitigating server overload are those of Twining et al. [23] and Tang et al. [21]. Twining et al. describe a prioritization mechanism that delays spam more than it delays legitimate mail. However, their problem is different, as they eventually accept all email, but just delay the spam. Such an approach would not work when all the mail simply cannot be accepted. While Tang et al. [21] do not consider the problem of server overload, they describe a mechanism to assign trust to and classify IP addresses using SVMs. Our work differs in the way it gets the historical reputations – rather than using a blackbox learning algorithm, it uses the IP addresses and network-aware clusters, thus directly utilizing the structure of the network.

There has also been interest in using reputation mechanisms for identifying spam. There are a few commercial IP-based reputation systems (e.g., SenderBase [2], TrustedSource [22]). A general reputation system for internet defense has been proposed in [9]. There has been work on using social network information for

designing reputation-granting mechanisms to mitigate spam [10, 11, 8]. Prakash et al. [18] propose community-based filters trained with classifiers to identify spam. Our work differs from these reputation systems as it demonstrates the potential of using network-aware clusters to assign reputations to IP addresses for prioritizing legitimate mail.

Recently, there have been studies on characterizing spammers, legitimate senders and mail traffic, and we only discuss the most closely related work here. Ramachandran and Feamster [19] present a detailed analysis of the network-level characteristics of spammers. By contrast, our work focuses on the comparison between legitimate mail and spam and explores the stability of legitimate mail. We also use network-aware clusters to probabilistically distinguish the bulk of the legitimate mail from the spam. Gomes et al. [12] study the e-mail arrivals, size distributions and temporal locality that distinguish spam traffic from non-spam traffic; these are interesting features that distinguish spam and legitimate traffic patterns and provide general insights into behaviour. Our measurement study differs as it focuses on understanding the historical behaviour of mail servers at the network level that can be exploited to practical spam mitigation.

## 6 Conclusion

In this paper, we have focused on using IP addresses as a computationally-efficient tool for spam mitigation in situations when the distinction need not be perfectly accurate. We performed an extensive analysis of IP addresses and network-aware clusters to identify properties that can distinguish the bulk of the legitimate mail and spam. Our analysis of IP addresses indicated that the bulk of the legitimate mail comes from long-lived IP addresses, while the analysis of network-aware clusters indicated that the bulk of the spam comes from clusters that are relatively long-lived. With these insights, we proposed and simulated a history-based reputation mechanism for prioritizing legitimate mail when the mail server is overloaded. Our simulations show that the history and the structure of the IP addresses can be used to substantially reduce the adverse impact of mail server overload on legitimate mail, by up to a factor of 3.

## 7 Acknowledgements

This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official poli-

cies or endorsements, either express or implied, of ARO, CMU, or the U.S. Government or any of its agencies. This material is also based upon work partially supported through the U.S. Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316, and by the National Science Foundation under Grants No. 0433540, 0448452 and CCF-0424422. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the ARO or the National Science Foundation. We thank Avrim Blum, Vyas Sekar and Elaine Shi for useful discussions and comments. We also thank the anonymous reviewers and our shepherd, David Dagon, for helpful comments on earlier versions of this paper.

## References

- [1] Brightmail. <http://www.brightmail.com>.
- [2] SenderBase. <http://www.senderbase.org>.
- [3] SpamAssassin. <http://www.spamassassin.org>.
- [4] SpamCop. <http://www.spamcop.net>.
- [5] SpamHaus. <http://www.spamhaus.net>.
- [6] AGARWAL, B., KUMAR, N., AND MOLLE, M. Controlling spam e-mails at the routers. In *IEEE International Conference on Communications (ICC)* (2005).
- [7] ANDROUTSOPOULOS, I., KOUTSIAS, J., CHANDRINOS, K., PALIOURAS, G., AND SPYROPOULOS, C. Spam filtering with Naive Bayes - which Naive Bayes? In *Third Conference on Email and Anti-Spam* (2006).
- [8] BOYKIN, P. O., AND ROYCHOWDHURY, V. P. Leveraging social networks to fight spam. *Computer* 38, 4 (2005), 61–68.
- [9] BRUMLEY, D., AND SONG, D. Towards attack-agnostic defenses. In *Proceedings of the First Workshop on Hot Topics in Security (HOTSEC)* (2006).
- [10] GARISS, S., KAMISKY, M., FREEDMAN, M., KARP, B., MAZIERES, D., AND YU, H. Re: Reliable email. In *Proceedings of NSDI* (2006).
- [11] GOLBECK, J., AND HENDLER, J. Reputation network analysis for e-mail filtering. In *First Conference on E-mail and Antispam* (2004).
- [12] GOMES, L. H., CAZITA, C., ALMEIDA, J. M., ALMEIDA, V., AND WAGNER MEIRA, J. Characterizing a spam traffic. In *Proceedings of Internet Measurement Conference (IMC)* (2004).
- [13] HARRIS, E. The next step in the spam control war: Greylisting. <http://projects.puremagic.com/greylisting/>.
- [14] JUNG, J., AND SIT, E. An empirical study of spam traffic and the use of DNS black lists. In *Proceedings of Internet Measurement Conference (IMC)* (2004).
- [15] KRISHNAMURTHY, B., AND WANG, J. On network-aware clustering of web clients. In *Proceedings of ACM SIGCOMM* (2000).
- [16] LOWD, D., AND MEEK, C. Good word attacks on statistical spam filters. In *Second Conference on Email and Anti-Spam* (2005).
- [17] MEDLOCK, B. An adaptive, semi-structured language model approach to spam filtering on a new corpus. In *Third Conference on Email and Anti-Spam* (2006).

- [18] PRAKASH, V. V., AND O'DONNELL, A. Fighting spam with reputation systems. *Queue* 3, 9 (2005), 36–41.
- [19] RAMACHANDRAN, A., AND FEAMSTER, N. Understanding the network-level behavior of spammers. In *Proceedings of ACM SIGCOMM* (2006).
- [20] SAHAMI, M., DUMAIS, S., HECKERMAN, D., AND HORVITZ, E. A Bayesian approach to filtering junk E-mail. In *Learning for Text Categorization: Papers from the 1998 Workshop* (1998), AAAI Technical Report WS-98-05.
- [21] TANG, Y., KRASSER, S., AND JUDGE, P. Fast and effective spam sender detection with granular SVM on highly imbalanced server behavior data. In *2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing* (2006).
- [22] TRUSTEDSOURCE. <http://www.trustedsource.org>.
- [23] TWINING, D., WILLIAMSON, M. M., MOWBRAY, M., AND RAHMOUNI, M. Email prioritization: Reducing delays on legitimate mail caused by junk mail. In *USENIX Annual Technical Conference* (2004).

## A Appendix

We present here the details of the policy used in Phase 1. for the history-based policies. In detail, the policy is the following: If the load is less than 75% of its capacity, the policy accepts all SMTP connections requests, regardless of the reputation of the IP address. If the load is greater than 75% of the capacity, the policy starts considering the reputation of the IP address and the legitimate mail that it expects to have to process in the near future.

For this purpose, it uses a distribution of the number of emails expected in the next  $t$  time units from reputation value at most  $k$  (for multiple  $k$  values), that is calculated based on the history of the distribution of mail arrival. Since our reputation function is the lifetime spam-ratio, a low reputation value is a good reputation, and a high reputation value is a bad reputation. Then it does the following: (a) given the current load, it computes the smallest  $k'$  such that all expected mail with reputations with  $k \leq k'$  can be processed on the server (b) it looks up the reputation of the IP address, and checks if it is higher than  $k'$ . (If the IP address does not have a known reputation value, and it does not belong to a cluster with a known reputation, then the IP address is assigned a relatively higher  $k'$  value. If  $k' \leq k$ , then the connection request of IP address is accepted, otherwise, it is rejected.

# BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation

Guofei Gu<sup>1</sup>, Phillip Porras<sup>2</sup>, Vinod Yegneswaran<sup>2</sup>, Martin Fong<sup>2</sup>, Wenke Lee<sup>1</sup>

<sup>1</sup>College of Computing  
Georgia Institute of Technology  
266 Ferst Drive  
Atlanta, GA 30332

<sup>2</sup>Computer Science Laboratory  
SRI International  
333 Ravenswood Avenue  
Menlo Park, CA 94025

## Abstract

We present a new kind of network perimeter monitoring strategy, which focuses on recognizing the infection and coordination dialog that occurs during a successful malware infection. BotHunter is an application designed to track the two-way communication flows between internal assets and external entities, developing an evidence trail of data exchanges that match a state-based infection sequence model. BotHunter consists of a correlation engine that is driven by three malware-focused network packet sensors, each charged with detecting specific stages of the malware infection process, including inbound scanning, exploit usage, egg downloading, outbound bot coordination dialog, and outbound attack propagation. The BotHunter correlator then ties together the dialog trail of inbound intrusion alarms with those outbound communication patterns that are highly indicative of successful local host infection. When a sequence of evidence is found to match BotHunter's infection dialog model, a consolidated report is produced to capture all the relevant events and event sources that played a role during the infection process. We refer to this analytical strategy of matching the dialog flows between internal assets and the broader Internet as *dialog-based correlation*, and contrast this strategy to other intrusion detection and alert correlation methods. We present our experimental results using BotHunter in both virtual and live testing environments, and discuss our Internet release of the BotHunter prototype. BotHunter is made available both for operational use and to help stimulate research in understanding the life cycle of malware infections.

## 1 Introduction

Over the last decade, malicious software or *malware* has risen to become a primary source of most of the scanning [38], (distributed) denial-of-service (DOS) activities [28], and direct attacks [5], taking place across the Internet. Among the various forms of malicious software, *botnets* in particular have recently distinguished

themselves to be among the premier threats to computing assets [20]. Like the previous generations of computer viruses and worms, a bot is a *self-propagating* application that infects vulnerable hosts through direct exploitation or Trojan insertion. However, all bots distinguish themselves from the other malware forms by their ability to establish a command and control (*C&C*) channel through which they can be updated and directed. Once collectively under the control of a *C&C* server, bots form what is referred to as a *botnet*. Botnets are effectively a collection of slave computing and data assets to be sold or traded for a variety of illicit activities, including information and computing resource theft, SPAM production, hosting phishing attacks, or for mounting distributed denial-of-service (DDoS) attacks [12, 34, 20].

Network-based intrusion detection systems (IDSs) and intrusion prevention systems (IPSs) may come to mind as the most appealing technology for detecting and mitigating botnet threats. Traditional IDSs, whether signature based [30, 35] or anomaly based [46, 8], typically focus on inbound packets flows for signs of malicious point-to-point intrusion attempts. Network IDSs have the capacity to detect initial incoming intrusion attempts, and the prolific frequency with which they produce such alarms in operational networks is well documented [36]. However, distinguishing a successful local host infection from the daily myriad of scans and intrusion attempts is as critical and challenging a task as any facet of network defense.

Intrusion report correlation enables an analyst to obtain higher-level interpretations of network sensor alert streams, thereby alleviating noise-level issues with traditional network IDSs. Indeed, there is significant research in the area of consolidating network security alarms into coherent incident pictures. One major vein of research in intrusion report correlation is that of alert fusion, *i.e.*, clustering similar events under a single label [42]. The primary goal of fusion is log reduction, and in most systems *similarity* is based upon either attributing multiple events to a single threat agent or providing a consoli-

dated view of a common set of events that target a single victim. The bot infection problem satisfies neither criterion. The bot infection process spans several diverse transactions that occur in multiple directions and potentially involves several active participants. A more applicable area of alert correlation research is multistage attack recognition, in which predefined scenario templates capture multiple state transition sequences that may be initiated by multiple threat agents [40, 29]. In Section 3 we discuss why predefined state transition models simply do not work well in bot infection monitoring. While we argue that bot infections do regularly follow a series of specific steps, we find it rare to accurately detect all steps, and find it equally difficult to predict the order and time-window in which these events are recorded.

**Our Approach:** We introduce an “evidence-trail” approach to recognizing successful bot infections through the communication sequences that occur during the infection process. We refer to this approach as the infection *dialog correlation* strategy. In dialog correlation, bot infections are modeled as a set of loosely ordered communication flows that are exchanged between an internal host and one or more external entities. Specifically, we model all bots as sharing a common set of underlying actions that occur during the infection life cycle: target scanning, infection exploit, binary egg download and execution, command and control channel establishment, and outbound scanning. We neither assume that all these events *are required* by all bots nor that every event *will be detected* by our sensor alert stream. Rather, our dialog correlation system collects an evidence trail of relevant infection events per internal host, looking for a threshold combination of sequences that will satisfy our requirements for bot declaration.

**Our System:** To demonstrate our methodology, we introduce a passive network monitoring system called *BotHunter*, which embodies our infection dialog correlation strategy. The *BotHunter* correlator is driven by Snort [35] with a customized malware-focused ruleset, which we further augment with two additional bot-specific anomaly-detection plug-ins for malware analysis: SLADE and SCADE. SLADE implements a lossy n-gram payload analysis of incoming traffic flows, targeting byte-distribution divergences in selected protocols that are indicative of common malware intrusions. SCADE performs several parallel and complementary malware-focused port scan analyses to both incoming and outgoing network traffic. The *BotHunter* correlator associates inbound scan and intrusion alarms with outbound communication patterns that are highly indicative of successful local host infection. When a sufficient sequence of alerts is found to match *BotHunter*’s infection dialog model, a consolidated report is produced to capture all the relevant events and event participants that

contributed to the infection dialog.

**Contributions:** Our primary contribution in this paper is to introduce a new network perimeter monitoring strategy, which focuses on detecting malware infections (specifically bots/botnets) through IDS-driven dialog correlation. We present an abstraction of the major network packet dialog sequences that occur during a successful bot infection, which we call our *bot infection dialog model*. Based on this model we introduce three bot-specific sensors, and our IDS-independent dialog correlation engine. Ours is the *first* real-time analysis system that can automatically derive a profile of the entire bot detection process, including the identification of the victim, the infection agent, the source of the egg download, and the command and control center.<sup>1</sup> We also present our analysis of *BotHunter* against more than 2,000 recent bot infection experiences, which we compiled by deploying *BotHunter* both within a high-interaction honeynet and through a VMware experimentation platform using recently captured bots. We validate our infection sequence model by demonstrating how our correlation engine successfully maps the network traces of a wide variety of recent bot infections into our model.

The remainder of this paper is outlined as follows. In Section 2 we discuss the sequences of communication exchanges that occur during a successful bot and worm infection. Section 3 presents our bot infection dialog model, and defines the conditions that compose our detection requirements. Section 4 presents the *BotHunter* architecture, and Section 5 presents our experiments performed to assess *BotHunter*’s detection performance. Section 6 discusses limitations and future work, and Section 7 presents related work. Section 8 discusses our Internet release of the *BotHunter* system, and in Section 9 we summarize our results.

## 2 Understanding Bot Infection Sequences

Understanding the full complexity of the bot infection life cycle is an important challenge for future network perimeter defenses. From the vantage point of the network egress position, distinguishing successful bot infections from the continual stream of background exploit attempts requires an analysis of the two-way dialog flow that occurs between a network’s internal hosts and the Internet. On a well-administered network, the threat of a direct-connect exploit is limited by the extent to which gateway filtering is enabled. However, contemporary malware families are highly versatile in their ability to attack susceptible hosts through email attachments, infected P2P media, and drive-by-download infections.

<sup>1</sup>Our current system implements a classic bot infection dialog model. One can define new models in an XML configuration file and add new detection sensors. Our correlator is IDS-independent, flexible, and extensible to process new models without modification.

Furthermore, with the ubiquity of mobile laptops and virtual private networks (VPNs), direct infection of an internal asset need not necessarily take place across an administered perimeter router. Regardless of how malware enters a host, once established inside the network perimeter the challenge remains to identify the infected machine and remove it as quickly as possible.

For this present study, we focus on a rather narrow aspect of bot behavior. Our objective is to understand the sequence of network communications and data exchanges that occur between a victim host and other network entities. To illustrate the stages of a bot infection, we outline an infection trace from one example bot, a variant of the Phatbot (aka Gaobot) family [4]. Figure 1 presents a summary of communication exchanges that were observed during a local host Phatbot infection.

As with many common bots that propagate through remote exploit injection, Phatbot first (step 1) probes an address range in search of exploitable network services or responses from Trojan backdoors that may be used to enter and hijack the infected machine. If Phatbot receives a connection reply to one of the targeted ports on a host, it then launches an exploit or logs in to the host using a backdoor. In our experimental case, a Windows workstation replies to a 135-TCP (MS DCE/RPC) connection request, establishing a connection that leads to an immediate RPC buffer overflow (step 2). Once infected, the victim host is directed by an upload shell script to open a communication channel back to the attacker to download the full Phatbot binary (step 3). The bot inserts itself into the system boot process, turns off security software, probes the local network for additional NetBIOS shares, and secures the host from other malware that may be loaded on the machine. The infected victim next distinguishes itself as a bot by establishing a connection to a botnet C&C server, which in the case of Phatbot is established over an IRC channel (step 4). Finally, the newly infected bot establishes a listen port to accept new binary updates and begins scanning other external victims on behalf of the botnet (step 5).

### 3 Modeling the Infection Dialog Process

While Figure 1 presents an example of a specific bot, the events enumerated are highly representative of the life cycle phases that we encounter across the various bot families that we have analyzed. Our bot propagation model is primarily driven by an assessment of outward-bound communication flows that are indicative of behavior associated with botnet coordination. Where possible, we seek to associate such outbound communication patterns with observed inbound intrusion activity. However, this latter activity is not a requirement for bot declaration. Neither are incoming scan and exploit alarms sufficient to declare a successful malware infection, as we assume

that a constant stream of scan and exploit signals will be observed from the egress monitor.

We model an infection sequence as a composition of participants and a loosely ordered sequence of exchanges: Infection  $I = \langle A, V, C, V', E, \bar{D} \rangle$ , where  $A$  = Attacker,  $V$  = Victim,  $E$  = Egg Download Location,  $C$  = C&C Server, and  $V'$  = the Victim's next propagation target.  $\bar{D}$  represents an infection dialog sequence composed of bidirectional flows that cross the egress boundary. Our infection dialog  $\bar{D}$  is composed of a set of five potential dialog transactions ( $E1, E2, E3, E4, E5$ ), some subset of which may be observed during an instance of a local host infection:

- E1: External to Internal Inbound Scan
- E2: External to Internal Inbound Exploit
- E3: Internal to External Binary Acquisition
- E4: Internal to External C&C Communication
- E5: Internal to External Outbound Infection Scanning

Figure 2 illustrates our bot infection dialog model used for assessing bidirectional flows across the network boundary. Our dialog model is similar to the model presented by Rajab et al. in their analysis of 192 IRC bot instances [33]. However, the two models differ in ways that arise because of our specific perspective of egress boundary monitoring. For example, we incorporate early initial scanning, which is often a preceding observation that occurs usually in the form of IP sweeps that target a relatively small set of selected vulnerable ports. We also exclude DNS C&C lookups, which Rajab et al. [33] include as a consistent precursor to C&C coordination, because DNS lookups are often locally handled or made through a designated DNS server via internal packet exchanges that should not be assumed visible from the egress position. Further, we exclude local host modifications and internal network propagation because these are also events that are not assumed to be visible from the egress point. Finally, we include internal-to-external attack propagation, which Rajab et al. [33] exclude. While our model is currently targeted for passive network monitoring events, it will be straightforward to include localhost-based or DNS-server-based IDSs that can augment our dialog model.

Figure 2 is not intended to provide a strict ordering of events, but rather to capture a typical infection dialog (exceptions to which we discuss below). In the idealized sequence of a direct-exploit bot infection dialog, the bot infection begins with an external-to-internal communication flow that may encompass bot scanning (E1) or a direct inbound exploit (E2). When an internal host has been successfully compromised (we observe that many compromise attempts regularly end with process dumps or system freezes), the newly compromised host down-

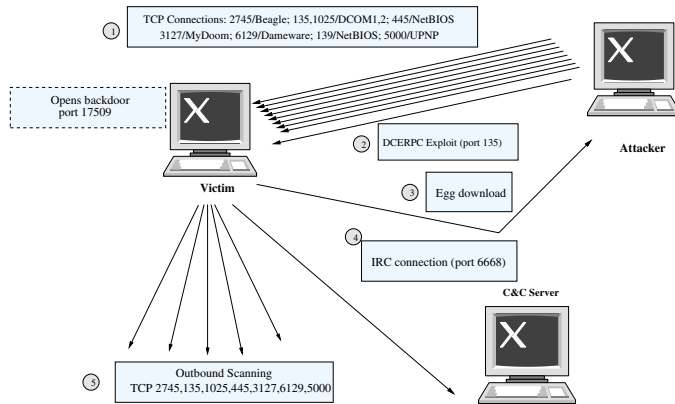


Figure 1: Phatbot Dialog Summary

loads and instantiates a full malicious binary instance of the bot (E3). Once the full binary instance of the bot is retrieved and executed, our model accommodates two potential dialog paths, which Rajab et al. [33] refer to as the bot Type I versus Type II split. Under Type II bots, the infected host proceeds to *C&C* server coordination (E4) before attempting self-propagation. Under a Type I bot, the infected host immediately moves to outbound scanning and attack propagation (E5), representing a classic worm infection.

We assume that bot dialog sequence analysis must be robust to the absence of some dialog events, must allow for multiple contributing candidates for each of the various dialog phases, and must not require strict sequencing on the order in which outbound dialog is conducted. Furthermore, in practice we have observed that for Type II infections, time delays between the initial infection events (E1 and E2) and subsequent outbound dialog events (E3, E4, and E5) can be significant—on the order of several hours. Furthermore, our model must be robust to failed E1 and E2 detections, possibly due to insufficient IDS fidelity or due to malware infections that occur through avenues other than direct remote exploit.

One approach to addressing the challenges of sequence order and event omission is to use a weighted event threshold system that captures the minimum necessary and sufficient sparse sequences of events under which bot profile declarations can be triggered. For example, one can define a weighting and threshold scheme for the appearance of each event such that a minimum set of event combinations is required before bot detection. In our case, we assert that bot infection declaration requires a minimum of

**Condition 1:** Evidence of local host infection (E2), AND evidence of outward bot coordination or attack propagation (E3-E5); or

**Condition 2:** At least two distinct signs of outward bot coordination or attack propagation (E3-E5).

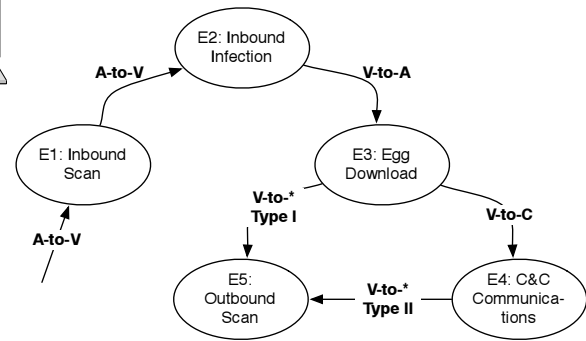


Figure 2: Bot Infection Dialog Model

In our description of the BotHunter correlation engine in Section 4, we discuss a weighted event threshold scheme that enforces the above minimum requirement for bot declaration.

## 4 BotHunter: System Design

We now turn our attention to the design of a passive monitoring system capable of recognizing the bidirectional warning signs of local host infections, and correlating this evidence against our dialog infection model. Our system, referred to as BotHunter, is composed of a trio of IDS components that monitor in- and out-bound traffic flows, coupled with our dialog correlation engine that produces consolidated pictures of successful bot infections. We envision BotHunter to be located at the boundary of a network, providing it a vantage point to observe the network communication flows that occur between the network's internal hosts and the Internet. Figure 3 illustrates the components within the BotHunter package.

Our IDS detection capabilities are composed on top of the open source release of Snort [35]. We take full advantage of Snort's signature engine, incorporating an extensive set of malware-specific signatures that we developed internally or compiled from the highly active Snort community (e.g., [10] among other sources). The signature engine enables us to produce dialog warnings for inbound exploit usage, egg downloading, and *C&C* patterns, as discussed in Section 4.1.3. In addition, we have developed two custom plugins that complement the Snort signature engine's ability to produce certain dialog warnings. Note that we refer to the various IDS alarms as *dialog warnings* because we do not intend the individual alerts to be processed by administrators in search of bot or worm activity. Rather, we use the alerts produced by our sensors as input to drive a bot dialog correlation analysis, the results of which are intended to capture and report the actors and evidence trail of a complete bot infection sequence.

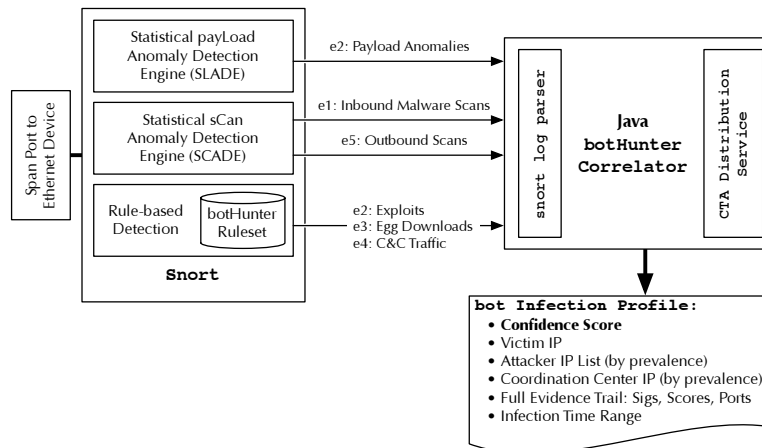


Figure 3: BotHunter System Architecture

Our two custom BotHunter plugins are called SCADE and SLADE. SCADE, discussed in Section 4.1.1, provides inbound and outbound scan detection warnings that are weighted for sensitivity toward malware-specific scanning patterns. SLADE, discussed in Section 4.1.2, conducts a byte-distribution payload anomaly detection of inbound packets, providing a complementary non-signature approach in inbound exploit detection.

Our BotHunter correlator is charged with maintaining an assessment of all dialog exchanges, as seen through our sensor dialog warnings, between all local hosts communicating with external entities across the Internet. The BotHunter correlator manages the state of all dialog warnings produced per local host in a data structure we refer to as the *network dialog correlation matrix* (Figure 4). Evidence of local host infection is evaluated and expired from our correlator until a sufficient combination of dialog warnings (E1–E5) crosses a weighted threshold. When the bot infection threshold is crossed for a given host, we produce a bot infection profile (illustrated in Figure 7).

Finally, our correlator also incorporates a module that allows users to report bot infection profiles to a remote repository for global collection and evaluation of bot activity. For this purpose, we utilize the Cyber-TA privacy-enabled alert delivery infrastructure [32]. Our delivery infrastructure first anonymizes all source-local addresses reported within the bot infection profile, and then delivers the profile to our data repository through a TLS over TOR [15] (onion routing protocol) network connection. These profiles will be made available to the research community, ideally to help in the large-scale assessment of bot dialog behavior, the sources and volume of various bot infections, and for surveying where *C&C* servers and exploit sources are located.

## 4.1 A Multiple-Sensor Approach to Gathering Infection Evidence

### 4.1.1 SCADE: Statistical sCan Anomaly Detection Engine

Recent measurement studies suggest that modern bots are packaged with around 15 exploit vectors on average [33] to improve opportunities for exploitation. Depending on how the attack source scans its target, we are likely to encounter some failed connection attempts prior to a successful infection.

To address this form aspect of malware interaction, we have designed SCADE, a Snort preprocessor plugin with two modules, one for inbound scan detection (E1 dialog warnings) and another for detecting outbound attack propagations (E5 dialog warnings) once our local system is infected. SCADE E1 alarms provide a potential early bound on the start of an infection, should this scan eventually lead to a successful infection.

**Inbound Scan Detection:** SCADE is similar in principle to existing scan detection techniques like [35, 24]. However, SCADE has been specifically weighted toward the detection of scans involving the ports often used by malware. It is also less vulnerable to DoS attacks because its memory trackers do not maintain per-source-IP state. Similar to the scan detection technique proposed in [48], SCADE tracks only scans that are specifically targeted to internal hosts, bounding its memory usage to the number of inside hosts. SCADE also bases its E1 scan detection on failed connection attempts, further narrowing its processing. We define two types of ports: *HS* (high-severity) ports representing highly vulnerable and commonly exploited services (e.g., 80/HTTP, 135, 1025/DCOM, 445/NetBIOS, 5000/UPNP, 3127/MyDoom) and *LS* (low-severity) ports.<sup>2</sup> Currently, we define

<sup>2</sup>Based on data obtained by analyzing vulnerability reports, mal-

26 TCP and 4 UDP HS ports and mark all others as LS ports. We set different weights to a failed scan attempt to different types of ports. An E1 dialog warning for a local host is produced based on an anomaly score that is calculated as  $s = w_1 F_{hs} + w_2 F_{ls}$ , where  $F_{hs}$  and  $F_{ls}$  indicate numbers of cumulative failed attempts at high-severity and low-severity ports, respectively.

**Outbound Scan Detection:** SCADE's outbound scan detection coverage for E5 dialog warnings is based on a voting scheme (AND, OR or MAJORITY) of three parallel anomaly detection models that track all external outbound connections per internal host:

- *Outbound scan rate* ( $s_1$ ): Detects local hosts that conduct high-rate scans across large sets of external addresses.
- *Outbound connection failure rate* ( $s_2$ ): Detects abnormally high connection fail rates, with sensitivity to HS port usage. We calculate the anomaly score  $s_2 = (w_1 F_{hs} + w_2 F_{ls})/C$ , where  $C$  is the total number of scans from the host within a time window.
- *Normalized entropy of scan target distribution* ( $s_3$ ): Calculates a Zipf (power-law) distribution of outbound address connection patterns [3]. A uniformly distributed scan target pattern provides an indication of a potential outbound scan. We use an anomaly scoring technique based on normalized entropy to identify such candidates:  $s_3 = \frac{H}{\ln(m)}$ , where the entropy of scan target distribution is  $H = -\sum_{i=1}^m p_i \ln(p_i)$ ,  $m$  is the total number of scan targets, and  $p_i$  is the percentage of the scans at target  $i$ . Each anomaly module issues a subalert when  $s_i \geq t_i$ , where  $t_i$  is a threshold. SCADE then uses a user-configurable "voting scheme", i.e., AND, OR, or MAJORITY, to combine the alerts from the three modules. For example, the AND rule dictates that SCADE issues an alert when all three modules issue an alert.

#### 4.1.2 SLADE: Statistical Payload Anomaly Detection Engine

SLADE is an anomaly-based engine for payload exploit detection. It examines the payload of every request packet sent to monitored services and outputs an alert if its lossy n-gram frequency deviates from an established normal profile.

SLADE is similar to PAYL [46], which is a payload-based 1-gram byte distribution anomaly detection scheme. PAYL examines the 1-gram byte distribution of the packet payload, i.e., it extracts 256 features each representing the occurrence frequency of one of the 256 possible byte values in the payload. A normal profile for a service/port, e.g., HTTP, is constructed by calculating the average and standard deviation of the feature vector of the normal traffic to the port. PAYL

calculates deviation distance of a test payload from the normal profile using a simplified Mahalanobis distance,  $d(x, y) = \sum_{i=0}^{255} (|x_i - y_i|) / (\sigma_i + \alpha)$ , where  $y_i$  is the mean,  $\sigma_i$  is the standard deviation, and  $\alpha$  is a smoothing factor. A payload is considered as anomalous if this distance exceeds a predetermined threshold. PAYL is effective in detecting worm exploits with a reasonable false positive rate as shown in [46, 47]. However, it could be evaded by a polymorphic blending attack (PBA) [18]. As discussed in [47, 18, 31], a generic n-gram version of PAYL may help to improve accuracy and the hardness of evasion. The n-grams extract n-byte sequence information from the payload, which helps in constructing a more precise model of the normal traffic compared to the single-byte (i.e., 1-gram) frequency-based model. In this case the feature space in use is not 256, but  $256^n$  dimensional. It is impractical to store and compute in a  $256^n$  dimension space for high-n-grams.

SLADE makes the n-gram scheme practical by using a lossy structure while still maintaining approximately the same accuracy as the original full n-gram version. We use a fixed vector counter (with size  $v$ ) to store a lossy n-gram distribution of the payload. When processing a payload, we sequentially scan n-gram substring  $str$ , apply some universal hash function  $h()$ , and increment the counter at the vector space indexed by  $h(str) \bmod v$ . We then calculate the distribution of the hashed n-gram indices within this (much) smaller vector space  $v$ . We define  $F$  as the feature space of n-gram PAYL (with a total of  $256^n$  distinct features), and  $F'$  as the feature space of SLADE (with  $v$  features).

This hash function provides a mapping from  $F$  to  $F'$  that we utilize for space efficiency. We require only  $v$  (e.g.,  $v = 2,000$ ), whereas n-gram PAYL needs  $256^n$  (e.g., even for a small  $n=3$ ,  $256^3 = 2^{24} \approx 16M$ ). The computational complexity in examining each payload is still linear ( $O(L)$ , where  $L$  is the length of payload), and the complexity in calculating distance is  $O(v)$  instead of  $256^n$ . Thus, the runtime performance of SLADE is comparable to 1-gram PAYL. Also note that although both use hashing techniques, SLADE is different from Anagram [45], which uses a Bloom filter to store all n-gram substrings from normal payloads. The hash function in SLADE is for feature compression and reduction, however the hash functions in Anagram are to reduce the false positives of string lookup in Bloom filter. In essence, Anagram is like a content matching scheme. It builds a huge knowledge base of all known good n-gram substrings using efficient storage and query optimizations provided by bloom filters, and examines a payload to determine whether the number of its n-gram substrings not in the knowledge base exceeds a threshold.

A natural concern of using such a lossy data structure is the issue of accuracy: how many errors (false pos-

ware infection vectors and analysis reports of datasets collected at Dshield.org and other honeynets.

itives and false negatives) may be introduced because of the lossy representation? To answer this question, we perform the following simple analysis.<sup>3</sup> Let us first overview the reason why the original n-gram PAYL can detect anomalies. We use  $\gamma$  to represent the number of non-zero value features in  $\mathbf{F}$  for a normal profile used by PAYL. Similarly,  $\gamma'$  is the number of non-zero value features in  $\mathbf{F}'$  for a normal profile used by SLADE. For a normal payload of  $length = L$ , there is a total of  $l = (L - n + 1)$  n-gram substrings. Among these  $l$  substrings,  $1 - \beta_n$  percent substrings converge to  $\gamma$  distinct features in the normal profile, *i.e.*, these substrings share similar distributions as the normal profile. The remaining (small portion)  $\beta_n$  percent of substrings are considered as noise substrings that do not belong to the  $\gamma$  features in the normal profile. For a malicious payload, if it can be detected as an anomaly, it should have a much larger portion of noise substrings  $\beta_a$  ( $\beta_a > \beta_n$ ).

We first analyze the false positives when using the lossy structure representation to see how likely SLADE will detect a normal (considered normal by n-gram PAYL) payload as anomalous. For a normal payload, the hashed indices of a  $1 - \beta_n$  portion of substrings (that converge to  $\gamma$  distinct features in  $\mathbf{F}$  for the normal profile of PAYL) should now converge in the new vector space (into  $\gamma'$  distinct features in  $\mathbf{F}'$  for the normal profile of SLADE). Because of the universal hash function, hashed indices of the  $\beta_n$  portion of noise substrings are most likely uniformly distributed into  $\mathbf{F}'$ . As a result, some of the original noise substrings may actually be hashed to the  $\gamma'$  distinct features in the normal profile of SLADE (*i.e.*, they may not be noise in the new feature space now). Thus, the deviation distance (*i.e.*, the anomaly score) can only decrease in SLADE. Hence, we conclude that SLADE may not have a higher false positive rate than n-gram PAYL.

Now let us analyze the false negative rate, *i.e.*, the likelihood that SLADE will treat a malicious payload (as would be detected by n-gram PAYL) as normal. False negatives happen when the hash collisions in the lossy structure mistakenly map a  $\beta_a$  portion of noise substrings into the  $\gamma'$  features (*i.e.*, the normal profile) for SLADE. By using the universal hash function, the probability for a noise substring to fall into  $\gamma'$  out of  $v$  space is  $\frac{\gamma'}{v}$ . Thus, the probability for all the  $l\beta_a$  noise substrings to collide into the  $\gamma'$  portion is about  $(\frac{\gamma'}{v})^{l\beta_a}$ . For example, if we assume  $v = 2,000$ ,  $\gamma' = 200$ ,  $l\beta_a = 100$ , then this probability is about  $(200/2000)^{100} = 1e - 100 \approx 0$ . In practice, the probability of such collisions for partial noise substrings is negligible. Thus, we believe that SLADE does not incur a significant accuracy penalty compared to

full n-gram PAYL, while significantly reducing its storage and computation complexity.

We measured the performance of SLADE in comparison to 1-gram PAYL by using the same data set as in [31]. The training and test data sets used were from the first and following four days of HTTP requests from the Georgia Tech campus network, respectively. The attack data consists of 18 HTTP-based buffer overflow attacks, including 11 regular (nonpolymorphic) exploits, 6 mimicry exploits generated by CLET, and 1 polymorphic blending attack used in [18] to evade 2-gram PAYL. In our experiment, we set  $n = 4$ ,  $v = 2,048$ .<sup>4</sup>

Table 1 summarizes our experimental results. Here, DFP is the desired false positive rate, *i.e.*, the rejection rate in the training set. RFP is the “real” false positive rate in our test data set. The detection rate is measured on the attack data set and is defined as the number of attack packets classified as anomalous divided by the total number of packets in the attack instances. We conclude from the results that SLADE performs better with respect to both DFP and RFP than the original PAYL (1-gram) system. Furthermore, we discovered that the minimum RFP for which PAYL is able to detect all attacks, including the polymorphic blending attack, is 4.02%. This is usually considered intolerably high for network intrusion detection. On the other hand, the minimum RFP required for SLADE to detect all attacks is 0.3601%. As shown in [31], 2-gram PAYL does not detect the polymorphic blending attack even if we are willing to tolerate an RFP as high as 11.25%. This is not surprising given that the polymorphic blending attack we used was specifically tailored to evade 2-gram PAYL. We also find that SLADE is comparable to (or even better than) a well-constructed ensemble IDS that combines 11 one-class SVM classifiers [31], and detects all the attacks, including the polymorphic blending attack, for an RFP at around 0.49%. SLADE also has the added advantage of more efficient resource utilization, which results in shorter training and execution times when compared to the ensemble IDS.

#### 4.1.3 Signature Engine: Bot-Specific Heuristics

Our final sensor contributor is the Snort signature engine. This module plays a significant role in detecting several of the classes of dialog warnings from our bot infection dialog model. Snort is our second sensor source for direct exploit detection (class E2), and our primary source for binary downloading (E3) and C&C communications (E4). We organize the rules selected for BotHunter into four separate rule files, covering 1046 E2 rules, 71 E3 rules, 246 E4 rules, and a small collection of 20 E5 rules, for total of 1383 heuristics. The rules are primarily de-

<sup>3</sup>We consider our analysis not as an exact mathematical proof, but an analytical description about the intuition behind SLADE.

<sup>4</sup>One can also choose a random  $v$  to better defeat evasion attacks like PBA. Also one may use multiple different hash functions and vectors for potential better accuracy and hardness of evasion.

Table 1: Performance of 1-gram PAYL and SLADE

	DFP(%)	0.0	0.01	0.1	1.0	2.0	5.0	10.0
<b>PAYL</b>	RFP(%)	0.00022	0.01451	0.15275	0.92694	1.86263	5.69681	11.05049
	Detected Attacks	1	4	17	17	17	18	18
	Detection Rate(%)	0.8	17.5	69.1	72.2	72.2	73.8	78.6
<b>SLADE</b>	RFP(%)	0.0026	0.0189	0.2839	1.9987	3.3335	6.3064	11.0698
	Detected Attacks	3	13	17	18	18	18	18
	Detection Rate(%)	20.6	74.6	92.9	99.2	99.2	99.2	99.2

rived from the Bleeding-Edge [10] and SourceFire’s registered free rulesets.

All the rulesets were selected specifically for their relevance to malware identification. Our rule selections are continually tested and reviewed across operational networks and our live honeynet environment. It is typical for our rule-based heuristics to produce less than 300 dialog warnings per 10-day period monitoring an operational border switch space port of approximately 130 operational hosts (SRI Computer Science Laboratory).

Our E2 ruleset focuses on the full spectrum of external to internal exploit injection attacks, and has been tested and augmented with rules derived from experimentation in our medium and high interactive honeynet environment, where we can observe and validate live malware infection attempts. Our E3 rules focus on (malware) executable download events from external sites to internal networks, covering as many indications of (malicious) binary executable downloads and download acknowledgment events as are in the publicly available Snort rulesets. Our E4 rules cover internally-initiated bot command and control dialog, and acknowledgment exchanges, with a significant emphasis on IRC and URL-based bot coordination.<sup>5</sup> Also covered are commonly used Trojan backdoor communications, and popular bot commands built by keyword searching across common major bot families and their variants. A small set of E5 rules is also incorporated to detect well-known internal to external backdoor sweeps, while SCADE provides the more in-depth hunt for general outbound port scanning.

#### 4.2 Dialog-Based IDS Correlation Engine

The BotHunter correlator tracks the sequences of IDS dialog warnings that occur between each local host and those external entities involved in these dialog exchanges. Dialog warnings are tracked over a temporal window, where each contributes to an overall infection sequence score that is maintained per local host. We introduce a data structure called the *network dialog correlation matrix*, which is managed, pruned, and evaluated by our correlation engine at each dialog warning insertion point. Our correlator employs a weighted threshold scoring function that aggregates the weighted scores

of each dialog warning, declaring a local host infected when a minimum combination of dialog transactions occur within our temporal pruning interval.

Figure 4 illustrates the structure of our *network dialog correlation matrix*. Each dynamically-allocated row corresponds to a summary of the ongoing dialog warnings that are raised between an individual local host and other external entities. The BotHunter correlator manages the five classes of dialog warnings presented in Section 3 (E1 through E5), and each event cell corresponds to one or more (possibly aggregated) sensor alerts that map into one of these five dialog warning classes. This correlation matrix dynamically grows when new activity involving a local host is detected, and shrinks when the observation window reaches an interval expiration.

In managing the dialog transaction history we employ an interval-based pruning algorithm to remove old dialog from the matrix. In Figure 4, each dialog may have one or two expiration intervals, corresponding to a *soft prune timer* (the open-faced clocks) and a *hard prune timer* (the filled clocks). The hard prune interval represents a fixed temporal interval over which dialog warnings are allowed to aggregate, and the end of which results in the calculation of our threshold score. The soft prune interval represents a smaller temporal window that allows users to configure tighter pruning interval requirements for high-production dialog warnings (inbound scan warnings are expired more quickly by the soft prune interval), while the others are allowed to accumulate through the hard prune interval. If a dialog warning expires solely because of a soft prune timer, the dialog is summarily discarded for lack of sufficient evidence (an example is row 1 in Figure 4 where only E1 has alarms). However, if a dialog expires because of a hard prune timer, the dialog threshold score is evaluated, leading either to a bot declaration or to the complete removal of the dialog trace should the threshold score be found insufficient.

To declare that a local host is infected, BotHunter must compute a sufficient and minimum threshold of evidence (as defined in Section 3) within its pruning interval. BotHunter employs two potential criteria required for bot declaration: 1) an incoming infection warning (E2) followed by outbound local host coordination or exploit propagation warnings (E3-E5), or 2) a minimum of

<sup>5</sup>E4 rules are essentially protocol, behavior and payload content signature, instead of a hard-coded known C&C domain list.

Int. Host	Timer	E1 $\ominus$	E2	E3	E4	E5
192.168.12.1	$\ominus$	$A_a \dots A_b$				
192.168.10.45	$\bullet$		$A_c \dots A_d$		$A_e \dots A_f$	
192.168.10.66	$\bullet$		$A_g$			
192.168.12.46	$\bullet$				$A_h \dots A_i$	$A_j \dots A_k$
:						
192.168.11.123	$\ominus \bullet$	$A_l$	$A_m \dots A_n$	$A_o$		

Figure 4: BotHunter Network Dialog Correlation Matrix

at least two forms of outbound bot dialog warnings (E3-E5). To translate these requirements into a scoring algorithm we employ a regression model to estimate dialog warning weights and a threshold value, and then test our values against a corpus of malware infection traces. We define an expectation table of predictor variables that match our conditions and apply a regression model where the estimated regression coefficients are the desired weights shown in Table 2. For completeness, the computed expectation table is provided in the project website [1].

	Coefficients	Standard Error
E1	0.09375	0.100518632
E2 rulebase	0.28125	0.075984943
E2 slade	0.09375	0.075984943
E3	0.34375	0.075984943
E4	0.34375	0.075984943
E5	0.34375	0.075984943

Table 2: Initial Weighting

These coefficients provide an approximate weighting system to match the initial expectation table <sup>6</sup>. We apply these values to our expectation table data to establish a threshold between bot and no-bot declaration. Figure 5 illustrates our results, where bot patterns are at X-axis value 1, and non-bot patterns are at X-axis 0. Bot scores are plotted vertically on the Y-axis. We observe that all but one non-bot patterns score below 0.6, and all but 2 bot patterns score above 0.65. Next, we examine our scoring model against a corpus of BotHunter IDS warning sets produced from successful bot and worm infections captured in the SRI honeynet between March and April 2007. Figure 6 plots the actual bot scores produced from these real bot infection traces. All observations produce BotHunter scores of 0.65 or greater.

When a dialog sequence is found to cross the threshold for bot declaration, BotHunter produces a *bot profile*. The bot profile represents a full analysis of roles

<sup>6</sup>In our model, we define E1 scans and the E2 anomaly score (produced by Slade) as increasers to infection confidence, such that our model lowers their weight influence.

of the dialog participants, summarizes the dialog alarms based on which dialog classes (E1-E5) the alarms map, and computes the infection time interval. Figure 7 (right) provides an example of a bot profile produced by the BotHunter correlation engine. The bot profile begins with an overall dialog anomaly score, followed by the IP address of the infected target (the victim machine), infector list, and possible *C&C* server. Then it outputs the dialog observation time and reporting time. The raw alerts specific to this dialog are listed in an organized (E1-E5) way and provide some detailed information.

## 5 Evaluating Detection Performance

To evaluate BotHunter's performance, we conducted several controlled experiments as well as real world deployment evaluations. We begin this section with a discussion of our detection performance while exposing BotHunter to infections from a wide variety of bot families using *in situ* virtual network experiments. We then discuss a larger set of true positive and false negative results while deploying BotHunter to a live VMWare-based high-interaction honeynet. This recent experiment exposed BotHunter to 2,019 instances of Windows XP and Windows 2000 direct-exploit malware infections from the Internet. We follow these controlled experiments with a brief discussion of an example detection experience using BotHunter during a live operational deployment.

Next, we discuss our broader testing experiences in two network environments. Here, our focus is on understanding BotHunter's daily false positive (FP) performance, at least in the context of two significantly different operational environments. A false positive in this context refers to the generation of a *bot profile* in response to a non-infection traffic flow, not to the number of IDS dialog warnings produced by the BotHunter sensors. As stated previously, network administrators are not expected to analyze individual IDS alarms. Indeed, we anticipate external entities to regularly probe and attack our networks, producing a regular flow of dialog warnings. Rather, we assert (and validate) that the dialog combinations necessary to cause a bot detection should

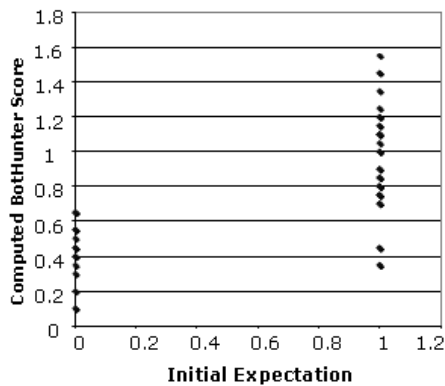


Figure 5: Scoring Plot from Expectation Table

be rarely encountered in normal operations.

### 5.1 Experiments in an *In situ* Virtual Network

Our evaluation setup uses a virtual network environment of three VMware guest systems. The first is a Linux machine with IRC server installed, which is used as the *C&C* server, and the other two are Windows 2000 instances. We infect one of the Windows instances and wait for it to connect to our *C&C* server. Upon connection establishment, we instruct the bot to start scanning and infecting neighboring hosts. We then await the infection and IRC *C&C* channel join by the second Windows instance. By monitoring the network activity of the second victim, we capture the full infection dialog. This methodology provides a useful means to measure the false negative performance of BotHunter.

We collected 10 different bot variants from three of the most well-known IRC-based bot families [20]: Agobot/Gaobot/Phatbot, SDBot/RBot/UrBot/UrXBot, and the mIRC-based GTbot. We then ran BotHunter in this virtual network and limited its correlation focus on the victim machine (essentially we assume the HOMENET is the victim’s IP). BotHunter successfully detected all bot infections (and produced bot profiles for all).

We summarize our measurement results for this virtual network infection experiment in Table 3. We use Yes or No to indicate whether a certain dialog warning is reported in the final profile. The two numbers within brackets are the number of generated dialog warnings in the whole virtual network and the number involving our victim, respectively. For example, for Phatbot-rls, 2,834 dialog warnings are generated by E2[rb] ([rb] means Snort rule base, [sl] means SLADE), but only 46 are relevant to our bot infection victim. Observe that although many warnings are generated by the sensors, only one bot profile is generated for this infection. This shows that BotHunter can significantly reduce the amount of information a security administrator needs to analyze. In our experiments almost all sensors worked as we expected. We do not see E1 events for RBot because the RBot fam-

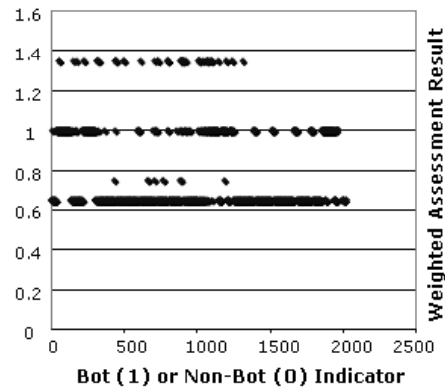


Figure 6: Scoring Plot: 2019 Real Bot Infections

ily does not provide any commands to trigger a vertical scan for all infection vectors (such as the “scan.startall” command provided by the Agobot/Phatbot family). The bot master must indicate a specific infection vector and port for each scan. We set our initial infection vector to DCOM, and since this was successful the attacking host did not attempt further exploits.

Note that two profiles are reported in the gt-with-dcom case. In the first profile, only E2[rb],E2[sl] and E4 are observed. In profile 2, E4 and E5 are observed (which is the case where we miss the initial infection periods). Because this infection scenario is very slow and lasts longer than our 4-minute correlation time window. Furthermore, note that we do not have any detected E3 dialog warnings reported for this infection sequence. Regardless, BotHunter successfully generates an infection profile. This demonstrates the utility of BotHunter’s evidence-trail-based dialog correlation model. We also reran this experiment with a 10-minute correlation time window, upon which BotHunter also reported a single infection profile.

### 5.2 SRI Honeynet Experiments

Our experimental honeynet framework has three integral components. The first component *Drone manager* is a software management component that is responsible for keeping track of drone availability and forwarding packets to various VMware instances. The address of one of the interfaces of this Intel Xeon 3 GHz dual core system is set to be the static route for the unused /17 network. The other interface is used for communicating with the high-interaction honeynet. Packet forwarding is accomplished using network address translation. One important requirements for this system is to keep track of infected drone systems and to recycle uninfected systems. Upon detecting a probable infection (outbound connections), we mark the drone as “tainted” to avoid reassigning that host to another source. Tainted drones are saved for manual analysis or automatically reverted back to previous clean snapshots after a fixed timeout. One

Table 3: Dialog Summary of Virtual Network Infections

	E1	E2[rb]	E2[sl]	E3	E4	E5
agobot3-priv4	Yes(2/2)	Yes(9/8)	Yes(6/6)	Yes(5)	Yes(38/8)	Yes(4/1)
phat-alpha5	Yes(14/4)	Yes(5,785/5,721)	Yes(6/2)	Yes(3/3)	Yes(28/26)	Yes(4/2)
phatbot-rls	Yes(11/3)	Yes(2,834/46)	Yes(6/2)	Yes(8/8)	Yes(69/20)	Yes(6/2)
rbot0.6.6	No(0)	Yes(2/1)	Yes(2/1)	Yes(2/2)	Yes(65/24)	Yes(2/1)
rxbot7.5	No(0)	Yes(2/2)	Yes(2/2)	Yes(2/2)	Yes(70/27)	Yes(2/1)
rx-asn-2-re-workedv2	No(0)	Yes(4/3)	Yes(3/2)	Yes(2/2)	Yes(59/18)	Yes(2/1)
Rxbot-ak-0.7-Modded.by.Uncanny	No(0)	Yes(3/2)	Yes(3/2)	Yes(2/2)	Yes(73/26)	Yes(2/1)
sxtbot6.5	No(0)	Yes(3/2)	Yes(3/2)	Yes(2/2)	Yes(65/24)	Yes(2/1)
Urx-Special-Ed-Ultra-2005	No(0)	Yes(3/2)	Yes(3/2)	Yes(2/2)	Yes(68/22)	Yes(2/1)
gt-with-dcom-profile1	No(1/0)	Yes(5/3)	Yes(6/2)	No(0)	Yes(221/1)	No(4/0)
gt-with-dcom-profile2	No(1/0)	No(5/0)	No(6/0)	No(0)	Yes(221/44)	Yes(4/2)
gt-with-dcom-10min-profile	No(1/0)	Yes(5/3)	Yes(6/3)	No(0)	Yes(221/51)	Yes(4/2)

of the interesting observations during our study was that most infection attempts did not succeed even on completely unpatched Windows 2000 and Windows XP systems. As a result, a surprisingly small number of VM instances was sufficient to monitor the sources contacting the entire /17 network. The second component is the *high-interaction-honeynet* system, which is hosted in a high-performance Intel Xeon 3 GHz dual core, dual CPU system with 8 GB of memory. For the experiments listed in this paper, we typically ran the system with 9 WinXP instances, 14 Windows 2000 instances (with two different service pack levels), and 3 Linux FC3 instances. The system was moderately utilized in this load. The final component is the *DNS/DHCP server*, which dynamically assigns IP addresses to VMware instances and also answers DNS queries from these hosts.

Over a 3-week period between March and April 2007, we analyzed a total of 2,019 successful WinXP and Win2K remote-exploit bot or worm infections. Each malware infection instance succeeded in causing the honeypot to initiate outbound communications related to the infection. Through our analysis of these traces using BotHunter sensor logs, we were able to very reliably observe the malware communications associated with the remote-to-local network service infection and the malware binary acquisition (egg download). In many instances we also observed the infected honeypot proceed to establish C&C communications and attempt to propagate to other victims in our honeynet. Through some of this experiment, our DNS service operated unreliably and some C&C coordination events were not observed due to DNS lookup failures.

Figure 7 illustrates a sample infection that was detected using the SRI honeynet, and the corresponding BotHunter profile. W32/IRCBot-TO is a very recent (released January 19, 2007) network worm/bot that propagates through open network shares and affects both Windows 2000 and Windows XP systems [37]. The worm uses the IPC share to connect to the `pipe` and leverages the MS06-40 exploit [27], which is a buffer

overflow that enables attackers to craft RPC requests that can execute arbitrary code. This mechanism is used to force the victim to fetch and execute a binary named `netadp.exe` from the system folder. The infected system then connects to the z3net IRC network and joins two channels upon which it is instructed to initiate scans of 203.0.0.0/8 network on several ports. Other bot families successfully detected by BotHunter included variants of W32/Korgo, W32/Virut.A and W32/Padobot.

Overall, BotHunter detected a total of 1,920 of these 2,019 successful bot infections. This represents a **95.1%** true positive rate. All malware instances observed during this period transmitted their exploits through ports TCP-445 or TCP-139. This is very common behavior, as the malware we observe tends to exploit the first vulnerable port that replies to a targeted scans, and ports TCP-445 and TCP-139 are usually among the first ports tested. The infection set analyzed exhibited limited diversity in the infection transmission methods, and overall we observed roughly 40 unique patterns in the dialog warnings produced.

This experiment produced 99 bot infections that *did not* produce bot profiles, *i.e.*, a **4.9%** false negative rate. To explain these occurrences we manually examined each bot infection trace that eluded BotHunter, using tcpdump and ethereal. The reasons for these failed bot detections can be classified into three primary categories: infection failures, honeynet setup or policy failures, or data corruption failures.

- *Infection failures:* We observed infections in which the exploit apparently led to instability and eventual failure in the infected host. More commonly, we observed cases in which the infected victim attempt to “phone home,” but the SYN request received no reply.

- *Honeynet setup and policy failures:* We observed that our NAT mechanism did not correctly translate application-level address requests (e.g., ftp PORT commands). This prevented several FTP egg download connection requests from proceeding, which would have otherwise led to egg download detections. In addition,

some traces were incomplete due to errors in our honeypot recycling logic which interfered with our observation of the infection logic.

- *Data corruption failures:* Data corruption was the dominant reason (86% of the failed traces) in preventing our BotHunter sensors from producing dialog warnings. We are still investigating the cause behind these corruptions, but suspect that these likely happened during log rotations by the Drone manager.

*Discussion:* In addition to the above false negative experiences, we also recognize that others reasons could prevent BotHunter from detecting infections. A natural extension of our *infection failures* is for a bot to purposely lay dormant once it has infected a host to avoid association of the infection transmission with an outbound egg download or coordination event. This strategy could be used successfully to circumvent BotHunter deployed with our default fixed pruning interval. While we found some infected victims failed to phone home, we could also envision the egg download source eventually responding to these requests after the BotHunter pruning interval, causing a similar missed association. Sensor coverage is of course another fundamental concern for any detection mechanism. Finally, while these results are highly encouraging, the honeynet environment provided a low-diversity in bot infections, in which attention was centered on direct exploits of TCP-445 and TCP-139. We did not provide a diversity of honeypots with various OSs, vulnerable services, or Trojan backdoors enabled, to fully examine the behavioral complexities of bots or worms.

### 5.3 An Example Detection in a Live Deployment

In addition to our laboratory and honeynet experiences, we have also fielded BotHunter to networks within the Georgia Tech campus network and within an SRI laboratory network. In the next sections we will discuss these deployments and our efforts to evaluate the false positive performance of BotHunter. First, we will briefly describe one example host infection that was detected using BotHunter within our Georgia Tech campus network experiments.

In early February 2007, BotHunter detected a bot infection that produced E1, E4 and E5 dialog warnings. Upon inspection of the bot profile, we observed that the bot-infected machine was scanned, joined an IRC channel, and began scanning other machines during the BotHunter time window. One unusual element in this experience was the omission of the actual infection transmission event (E2), which is observed with high-frequency in our live honeynet testing environment. We assert that the bot profile represents an actual infection because during our examination of this infection report, we discovered that the target of the E4 (C&C Server) dialog warn-

ing was an address that was blacklisted both by the ShadowServer and the botnet mailing list as a known C&C server during the time of our bot profile.

### 5.4 Experiments in a University Campus Network

In this experiment, we evaluate the detection and false positive performance of BotHunter in a production campus network (at the College of Computing [CoC] at Georgia Tech). The time period of this evaluation was between October 2006 and February 2007.

The monitored link exhibits typical diurnal behavior and a sustained peak traffic of over 100 Mbps during the day. While we were concerned that such traffic rates might overload typical NIDS rulesets and real-time detection systems, our experience shows that it is possible to run BotHunter live under such high traffic rates using commodity PCs. Our BotHunter instance runs on a Linux server with an Intel Xeon 3.6 GHz CPU and 6 GB of memory. The system runs with average CPU and memory utilization of 28% and 3%, respectively.

To evaluate the representativeness of this traffic, we randomly sampled packets for analysis (about 40 minutes). The packets in our sample, which were almost evenly distributed between TCP and UDP, demonstrated wide diversity in protocols, including popular protocols such as HTTP, SMTP, POP, FTP, SSH, DNS, and SNMP, and collaborative applications such as IM (e.g., ICQ, AIM), P2P (e.g., Gnutella, Edonkey, bittorrent), and IRC, which share similarities with infection dialog (e.g., two-way communication). We believe the high volume of background traffic, involving large numbers of hosts and a diverse application mix, offers an appealing environment to confirm our detection performance, and to examine the false positive question.

First, we evaluated the detection performance of BotHunter in the presence of significant background traffic. We injected bot traffic captured in the virtual network (from the experiments described in Section 5.1) into the captured Georgia Tech network traffic. Our motivation was to simulate real network infections for which we have the ground truth information. In these experiments, BotHunter correctly detected all 10 injected infections (by the 10 bots described in Section 5.1).

Next, we conducted a longer-term (4 months) evaluation of false alarm production. Table 4 summarizes the number of dialog warnings generated by BotHunter for each event type from October 2006 to January 2007. BotHunter sensors generated about 2,563,402 (more than 20,000 per day) raw dialog warnings from all the five event categories. For example, many E3 dialog warnings report on Windows executable downloads, which by themselves do not shed light on the presence of exploitable vulnerabilities. However, our experiments do demonstrate that the alignment of the bot detection con-

Table 4: Raw Alerts of BotHunter in 4 Month Operation in CoC Network

Event	E1	E2[rb]	E2[sl]	E3	E4	E5
Alert#	550,373	950,112	316,467	1,013	697,374	48,063

ditions outline in Section 3 rarely align within a stream of dialog warnings from normal traffic patterns. In fact, only 98 profiles were generated in 4 months, less than one per day on average.

In further analyzing these 98 profiles, we had the following findings. First, there are no false positives related to any normal usage of collaborative applications such as P2P, IM, or IRC. Almost two-thirds (60) of the bot profiles involved access to an MS-Exchange SMB server (33) and SMTP server (27). In the former case, the bot profiles described a NETBIOS SMB-DS IPC\$ unicode share access followed by a windows executable downloading event. Bleeding Edge Snort’s IRC rules are sensitive to some IRC commands (e.g., USER) that frequently appear in the SMTP header. These issues could easily be mitigated by additional whitelisting of certain alerts on these servers. The remaining profiles contained mainly two event types and with low overall confidence scores. Additional analysis of these incidents was complicated by the lack of full packet traces in our high-speed network. We can conservatively assume that they are false positives and thus our experiments here provide a reasonable estimate of the upper bound on the number of false alarms (less than one) in a busy campus network.

### 5.5 Experiments in an Institutional Laboratory

We deployed BotHunter live on a small well-administered production network (a lightly used /17 network that we can say with high confidence is infection free). Here, we describe our results from running BotHunter in this environment. Our motivation for conducting this experiment was to obtain experience with false positive production in an operational environment, where we could also track all network traces and fully evaluate the conditions that may cause the production of any unexpected bot profiles.

BotHunter conducted a 10-day data stream monitoring test from the span port position of an egress border switch. The network consists of roughly 130 active IP addresses, an 85% Linux-based host population, and an active user base of approximately 54 people. During this period, 182 million packets were analyzed, consisting of 152 million TCP packets (83.5%), 15.8 million UDP packets (8.7%), and 14.1 million ICMP packets (7.7%). Our BotHunter sensors produced 5,501 dialog warnings, composed of 1,378 E1 scan events, 20 E2 exploit signature events, 193 E3 egg-download signature events, 7 E4 C&C signature events and 3,904 E5 scan events. From these dialog warnings, the BotHunter correlator

produced just one bot profile. Our subsequent analysis of the packets that caused the bot profile found that this was a false alarm. Upon packet inspection, it was found that the session for which the bot declaration occurred consisted of a 1.6 GB multifile FTP transfer, during which a binary image was transferred with content that matched one of our buffer overflow detection patterns. The buffer overflow false alarm was coupled with a second MS Windows binary download, which caused BotHunter to cross our detection threshold and declare a bot infection.

## 6 Limitations and Future Work

Several important practical considerations present challenges in extending and adapting BotHunter for arbitrary networks in the future.

### Adapting to Emerging Threats and Adversaries:

Network defense is a perennial arms race<sup>7</sup> and we anticipate that the threat landscape could evolve in several ways to evade BotHunter. First, bots could use encrypted communication channels for C&C. Second, they could adopt more stealthy scanning techniques. However, the fact remains that hundreds of thousands of systems remain unprotected, attacks still happen in the clear, and adversaries have not been forced to innovate. Open-source systems such as BotHunter would raise the bar for successful infections. Moreover, BotHunter could be extended with anomaly-based “entropy detectors” for identification of encrypted channels. We have preliminary results that are promising and defer deeper investigation to future work. We are also developing new anomaly-based C&C detection schemes (for E4).

It is also conceivable that if BotHunter is widely deployed, adversaries would devise clever means to evade the system, e.g., by using attacks on BotHunter’s dialog history timers. One countermeasure is to incorporate an additional random delay to the hard prune interval, thereby introducing uncertainty into how long BotHunter maintains local dialog histories.

**Incorporating Additional State Logic:** The current set of states in the bot infection model was based on the behavior of contemporary bots. As bots evolve, it is conceivable that this set of states would have to be extended or otherwise modified to reflect the current threat landscape. This could be accomplished with simple configuration changes to the BotHunter correlator. We expect such changes to be fairly infrequent as they reflect fun-

<sup>7</sup>In this race, we consider BotHunter to be a substantial technological escalation for the white hats.

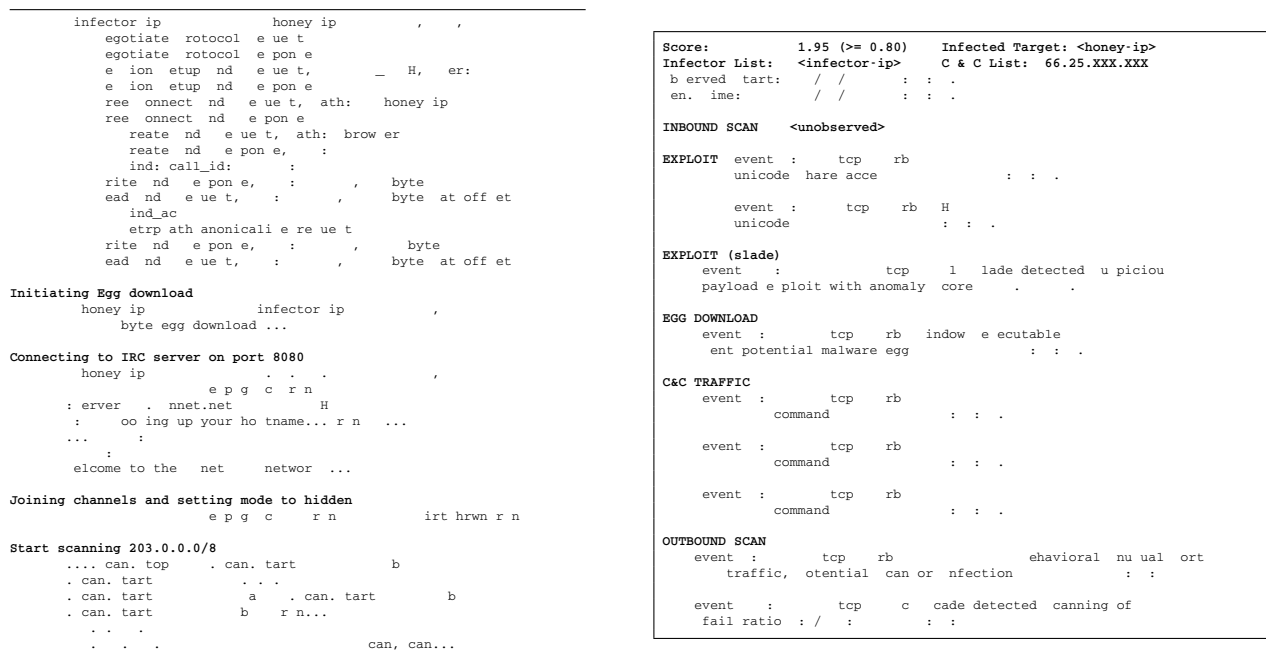


Figure 7: Honeynet Interaction Summary (left) and corresponding BotHunter Profile (right) for W32/IRCBot-TO

damental paradigm shifts in bot behavior.

## 7 Related Work

Recently, there has been a significant thrust in research on botnets. To date, the primary focus of much of this research has been on gaining a basic understanding of the nature and full potential of the botnet threat. Rajab et al. provided an in-depth study in understanding the dynamics of botnet behavior in the large, employing “longitudinal tracking” of IRC botnets through IRC and DNS tracking techniques [33]. Researchers have also studied the dynamics of botnet C&C protocols [19, 50], including global dynamics such as diurnal behavior [14]. Other studies have investigated the internals of bot instances to examine the structural similarities, defense mechanisms, and command and control capabilities of the major bot families [7] and developed techniques to automatically harvest malware samples directly from the Internet [6]. There is also some very recent work on the detection of botnets. Rishi [21] is an IRC botnet detection system that uses n-gram analysis to identify botnet nickname patterns. Binkley and Singh [9] proposed an anomaly based system that combines IRC statistics and TCP work weight for detecting IRC-based botnets. Livadas et al. [26] proposed a machine learning based approach for botnet detection. Karasaridis et al. [25] presented an algorithm for detecting IRC botnet controllers from net-flow records. These efforts are complementary in that they could provide additional BotHunter evidence-trails for infection events.

A significant amount of related work has investigated alert correlation techniques for network intrusion detection. An approach to capturing complex and multistep attacks is to explicitly specify the stages, relationships, and ordering among the various constituents of an attack. GridS [39] aggregates network activity into activity graphs that can be used for analyzing causal structures and identifying policy violations. CARDS is a distributed system for detecting and mitigating coordinated attacks [49]. Abad et al. [2] proposed to correlate data among different sources/logs (e.g., syslog, firewall, net-flow) to improve intrusion detection system accuracy. Ellis et al. and Jiang et al. describe two behavioral-based systems for detecting network worms [17, 23]. In contrast to the above systems, our work focuses on the problem of bot detection and uses infection dialog correlation as a means to define the probable set of events that indicate a bot infection.

Sommer et al. [36] describe contextual Bro signatures as a means for producing expressive signatures and weeding out false positives. These signatures capture two dialogs and are capable of precisely defining multistep attacks. Our work differs from this in our requirement to simultaneously monitor several flows across many participants (e.g., infection source, bot victim, C&C, propagation targets) and our evidence-trail-based approach to loosely specify bot infections.

JIGSAW is a system that uses notions of concepts and capabilities for modeling complex attacks [40] and [29] provides a formal framework for alert correla-

tion. CAML is a language framework for defining and detecting multistep attack scenarios [11]. Unlike BotHunter, all these systems are based on causal relationships *i.e.*, pre-conditions and post-conditions of attacks. An obvious limitation is that these dependencies, need to be manually specified a priori for all attacks, and yet such dependencies are often unknown.

Alert correlation modules such as CRIM [13] provide the ability to cluster and correlate similar alerts. The system has the capability to extract higher-level correlation rules automatically for the purpose of intention recognition. In [42], Valdes and Skinner propose a two-step probabilistic alert correlation based on attack threads and alert fusion. We consider this line of work to be complementary, *i.e.*, these fusion techniques could be integrated into the BotHunter framework as a preprocessing step in a multisensor environment.

USTAT [22] and NetSTAT [44] are two IDSs based on state transition analysis techniques. They specify computer attacks as sequences of actions that cause transitions in the security state of a system. In [43], multistep attack correlation is performed on attack scenarios specified (a priori) using STATL [16], which is a language for expressing attacks as states and transitions. Our work differs from these systems in that we do not have a strict requirement of temporal sequence, and can tolerate missing events during the infection flow.

## 8 BotHunter Internet Distribution

We are making BotHunter available as a free Internet distribution for use in testing and facilitating research with the hope that this initiative would stimulate community development of extensions.

A key component of the BotHunter distribution is the Java-based correlator that by default reads alert streams from Snort. We have tested our system with Snort 2.6.\* and it can be downloaded from [www.cyber-ta.org/botHunter/](http://www.cyber-ta.org/botHunter/). A noteworthy feature of the distribution is integrated support for “large-scale privacy-preserving data sharing”. Users can enable an option to deliver secure anonymous bot profiles to the Cyber-TA security repository [32], the collection of which we will make available to providers and researchers. The repository is currently operational and in beta release of its first report delivery software.

Our envisioned access model is similar to that of DShield.org [41] with the following important differences. First, our repository is blind to who is submitting the bot report and the system will deliver alerts via TLS over TOR, preventing an association of bot reports to a site via passive sniffing. Second, our anonymization strategy obfuscates all local IP addresses and time intervals in the profile database but preserves *C&C*, egg download, and attacker addresses that do not match user defined address proximity mask. Users can enable fur-

ther field anonymizations as they require. We intend to use contributed bot profiles to learn specific alert signature patterns for specific bots, to track attackers, and to identify *C&C* sites.

## 9 Conclusion

We have presented the design and implementation of BotHunter, a perimeter monitoring system for real-time detection of Internet malware infections. The cornerstone of the BotHunter system is a three-sensor dialog correlation engine that performs alert consolidation and evidence trail gathering for investigation of putative infections. We evaluate the system’s detection capabilities in an *in situ* virtual network and a live honeynet demonstrating that the system is capable of accurately flagging both well-studied and emergent bots. We also validate low false positive rates by running the system live in two operational production networks. Our experience demonstrates that the system is highly scalable and reliable (very low false positive rates) even with not-so-reliable (weak) raw detectors. BotHunter is also the *first* example of a widely distributed bot infection profile analysis tool. We hope that our Internet release will enable the community to extend and maintain this capability while inspiring new research directions.

## 10 Acknowledgements

We are thankful to Al Valdes for his help in developing the regression model for computing weights of dialog events. We thank Roberto Perdisci and Junjie Zhang for helpful discussions on the early version of this work. This material is based upon work supported through the U.S. Army Research Office (ARO) under the Cyber-TA Research Grant No.W911NF-06-1-0316 and Grant W911NF0610042, and by the National Science Foundation under Grants CCR-0133629 and CNS-0627477. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of U.S. ARO or the National Science Foundation.

## References

- [1] Cyber-TA: BotHunter distribution page. <http://www.cyber-ta.org/releases/botHunter/index.html>, 2007.
- [2] C. Abad, J. Taylor, C. Sengul, W. Yurcik, Y. Zhou, and K. Rowe. Log correlation for intrusion detection: A proof of concept. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC’03)*, page 255, Washington, DC, USA, 2003. IEEE Computer Society.
- [3] L. A. Adamic and B. A. Huberman. Zipf’s law and the Internet. *Glottometrics*, 3:143–150, 2002.
- [4] W32 Agobot IB. <http://www.sophos.com/virusinfo/analyses/trojagobotib.html>.
- [5] K. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. Keromytis. Detecting targeted attacks using shadow honeypots. In *Usenix Security Symposium*, Baltimore, Maryland, August 2005.

- [6] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling. The nepenthes platform: An efficient approach to collect malware. In *Proceedings of Recent Advances in Intrusion Detection*, Hamburg, September 2006.
- [7] P. Barford and V. Yegneswaran. An inside look at botnets. Special Workshop on Malware Detection, Advances in Information Security, Springer Verlag, 2006.
- [8] S. Biles. Detecting the unknown with snort and statistical packet anomaly detection engine (SPADE). <http://www.computersecurityonline.com/spade/SPADE.pdf>.
- [9] J. R. Binkley and S. Singh. An algorithm for anomaly-based botnet detection. In *Proceedings of USENIX Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI)*, pages 43–48, July 2006.
- [10] Bleeding Edge Threats. The Bleeding Edge of Snort. <http://www.bleedingsnort.com/>, 2007.
- [11] S. Cheung, M. Fong, and U. Lindqvist. Modeling multistep cyber attacks for scenario recognition. In *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX III)*, 2003.
- [12] E. Cooke, F. Jahanian, and D. McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Proceedings of Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI'05)*, 2005.
- [13] F. Cuppens and A. Mieke. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of IEEE Symposium on Security and Privacy*, 2002.
- [14] D. Dagon, C. Zou, and W. Lee. Modeling botnet propagation using timezones. In *Proceedings of Network and Distributed Security Symposium (NDSS '06)*, January 2006.
- [15] R. Dingledine, N. Mathewson, and P. Syverson. TOR: The second generation onion router. In *Proceedings of the Usenix Security Symposium*, 2004.
- [16] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. Statl: An attack language for state-based intrusion detection. *Journal of Computer Security*, 10, 2002.
- [17] D. R. Ellis, J. G. Aiken, K. S. Attwood, and S. D. Tenaglia. A Behavioral Approach to Worm Detection. In *Proceedings of WORM*, 2004.
- [18] P. Fogla, M. Sharif, R. Perdisci, O. M. Kolesnikov, and W. Lee. Polymorphic blending attack. In *Proceedings of the 2006 USENIX Security Symposium*, 2006.
- [19] F. Freiling, T. Holz, and G. Wicherski. Botnet tracking: Exploring a root-cause methodology to prevent denial of service attacks. In *Proceedings of ESORICS*, 2005.
- [20] German HoneyNet Project. Tracking botnets. <http://www.honeynet.org/papers/bots>, 2005.
- [21] J. Goebel and T. Holz. Rishi: Identify bot contaminated hosts by irc nickname evaluation. In *USENIX Workshop on Hot Topics in Understanding Botnets (HotBots'07)*, 2007.
- [22] K. Iglun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection system. *IEEE Transactions on Software Engineering*, 21, 1995.
- [23] X. Jiang and D. Xu. Profiling self-propagating worms via behavioral footprinting. In *Proceedings of CCS WORM*, 2006.
- [24] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy 2004*, Oakland, CA, May 2004.
- [25] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale botnet detection and characterization. In *USENIX Workshop on Hot Topics in Understanding Botnets (HotBots'07)*, 2007.
- [26] C. Livadas, R. Walsh, D. Lapsley, and W. T. Strayer. Using machine learning techniques to identify botnet traffic. In *2nd IEEE LCN Workshop on Network Security (WoNS'2006)*, 2006.
- [27] Mitre Corporation. CVE-2006-3439. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3439>.
- [28] D. Moore, G. Voelker, and S. Savage. Inferring Internet denial-of-service activity. In *Proceedings of the 10th USENIX Security Symposium*, 2001.
- [29] P. Ning, Y. Cui, and D. Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *Proceedings of Computer and Communications Security*, 2002.
- [30] V. Paxson. BRO: A System for Detecting Network Intruders in Real Time. In *Proceedings of the 7th USENIX Security Symposium*, 1998.
- [31] R. Perdisci, G. Gu, and W. Lee. Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems. In *Proceedings of the IEEE International Conference on Data Mining (ICDM'06)*, December 2006.
- [32] P. Porras. Privacy-enabled global threat monitoring. In *Proceedings of the IEEE Security and Privacy Magazine*, 2006.
- [33] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multi-faceted approach to understanding the botnet phenomenon. In *Proceedings of ACM SIGCOMM/USENIX Internet Measurement Conference*, Brazil, October 2006.
- [34] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *Proceedings of ACM SIGCOMM*, 2006.
- [35] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of USENIX LISA'99*, 1999.
- [36] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *10th ACM Conference on Computer and Communication Security (CCS)*, Washington, DC, October 2003.
- [37] Sophos Inc. W32/IRCBot-TO. <http://www.sophos.com/virusinfo/analyses/w32ircbotto.html>, 2007.
- [38] S. Staniford, J. Hoagland, and J. McAlerney. Practical automated detection of stealthy portscans. In *Journal of Computer Security*, 2002.
- [39] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. Grids—a graph based intrusion detection system for large networks. In *19th National Information Systems Security Conference*, 1996.
- [40] S. Templeton and K. Levitt. A requires/provides model for computer attacks. In *New Security Paradigms Workshop*, 2000.
- [41] J. Ullrich. DSHIELD. <http://www.dshield.org>, 2007.
- [42] A. Valdes and K. Skinner. Probabilistic alert correlation. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, pages 54–68, 2001.
- [43] F. Valeur, G. Vigna, C. Kruegel, and R. Kemmerer. Comprehensive approach to intrusion detection alert correlation. In *Proceedings of IEEE Transactions on Dependable and Secure Computing*, 2004.
- [44] G. Vigna and R. Kemmerer. NetSTAT: A network-based intrusion detection approach. In *Proceedings of the 14th Annual Computer Security Applications Conference (ACSAC '98)*, 1998.
- [45] K. Wang, J. J. Parekh, and S. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.
- [46] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.
- [47] K. Wang and S. Stolfo. Anomalous payload-based worm detection and signature generation. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.
- [48] D. Whyte, P. van Oorschot, and E. Kranakis. Exposure maps: Removing reliance on attribution during scan detection. In *Proceedings of 1st USENIX Workshop on Hot Topics in Security (HotSec'06)*, 2006.
- [49] J. Yang, P. Ning, X. S. Wang, and S. Jajodia. Cards: A distributed system for detecting coordinated attacks. In *SEC*, 2000.
- [50] V. Yegneswaran, P. Barford, and V. Paxson. Using honeynets for Internet situational awareness. In *Proceedings of the Fourth Workshop on Hot Topics in Networks (HotNets IV)*, College Park, MD, November 2005.

# Integrity Checking in Cryptographic File Systems with Constant Trusted Storage

Alina Oprea\*

Michael K. Reiter†

## Abstract

In this paper we propose two new constructions for protecting the integrity of files in cryptographic file systems. Our constructions are designed to exploit two characteristics of many file-system workloads, namely low entropy of file contents and high sequentiality of file block writes. At the same time, our approaches maintain the best features of the most commonly used algorithm today (Merkle trees), including defense against replay of stale (previously overwritten) blocks and a small, constant amount of trusted storage per file. Via implementations in the EncFS cryptographic file system, we evaluate the performance and storage requirements of our new constructions compared to those of Merkle trees. We conclude with guidelines for choosing the best integrity algorithm depending on typical application workload.

## 1 Introduction

The growth of outsourced storage in the form of storage service providers underlines the importance of developing efficient security mechanisms to protect files stored remotely. Cryptographic file systems (e.g., [10, 6, 25, 13, 17, 23, 20]) provide means to protect file secrecy (i.e., prevent leakage of file contents) and integrity (i.e., detect the unauthorized modification of file contents) against the compromise of the file store and attacks on the network while blocks are in transit to/from the file store. Several engineering goals have emerged to guide the design of efficient cryptographic file systems. First, cryptographic protections should be applied at the granularity of individual blocks as opposed to entire files, since the

latter requires the entire file to be retrieved to verify its integrity, for example. Second, applying cryptographic protections to a block should not increase the block size, so as to be transparent to the underlying block store. (Cryptographic protections might increase the number of blocks, however.) Third, the trusted storage required by clients (e.g., for encryption keys and integrity verification information) should be kept to a minimum.

In this paper we propose and evaluate two new algorithms for protecting file integrity in cryptographic file systems. Our algorithms meet these design goals, and in particular implement integrity using only a small constant amount of trusted storage per file. (Of course, as with any integrity-protection scheme, this trusted information for many files could itself be written to a file in the cryptographic file system, thereby reducing the trusted storage costs for many files to that of only one. The need for trusted information cannot be entirely eliminated, however.) In addition, our algorithms exploit two properties of many file-system workloads to achieve efficiencies over prior proposals. First, typical file contents in many file-system workloads have low empirical entropy; such is the case with text files, for example. Our first algorithm builds on our prior proposal that exploits this property [26] and uses tweakable ciphers [21, 15] for encrypting file block contents; this prior proposal, however, did not achieve constant trusted storage per file. Our second algorithm reduces the amount of additional storage needed for integrity by using the fact that low-entropy block contents can be compressed enough to embed a message-authentication code inside the block. The second property that we exploit in our algorithms to reduce the additional storage needed for integrity is that blocks of the same file are often written sequentially, a characteristic that, to our knowledge, has not been previously utilized.

\*Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA; [alina@cs.cmu.edu](mailto:alina@cs.cmu.edu)

†Electrical & Computer Engineering Department and Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA; [reiter@cmu.edu](mailto:reiter@cmu.edu)

By designing integrity mechanisms that exploit these properties, we demonstrate more efficient integrity protections in cryptographic file systems than have previously been possible for many workloads. The measures of efficiency that we consider include the amount of untrusted storage required by the integrity mechanism (over and above that required for file blocks); the *integrity bandwidth*, i.e., the amount of this information that must be accessed (updated or read) when accessing a single file block, averaged over all blocks in a file, all blocks in all files, or all accesses in a trace (depending on context); and the file write and read performance costs.

The standard against which we compare our algorithms is the Merkle tree [24], which to date is the overwhelmingly most popular method of integrity protection for a file. Merkle trees can be implemented in cryptographic file systems so as to meet the requirements outlined above, in particular requiring trusted storage per file of only one output of a cryptographic hash function (e.g., 20 bytes for SHA-1 [30]). They additionally offer an integrity bandwidth per file that is logarithmic in the number of file blocks. However, Merkle trees are oblivious to file block contents and access characteristics, and we show that by exploiting these, we can generate far more efficient integrity mechanisms for some workloads.

We have implemented our integrity constructions and Merkle trees in EncFS [14], an open-source user-level file system that transparently provides file block encryption on top of FUSE [12]. We provide an evaluation of the three approaches with respect to our measures of interest, demonstrating how file contents, as well as file access patterns, have a great influence on the performance of the new integrity algorithms. Our experiments demonstrate that there is not a clear winner among the three constructions for *all* workloads, in that different integrity constructions are best suited to particular workloads. We thus conclude that a cryptographic file system should implement all three schemes and give higher-level applications an option to choose the appropriate integrity mechanism.

## 2 Random Access Integrity Model

We consider the model of a cryptographic file system that provides random access to files. Encrypted data is stored on untrusted storage servers and there is a mechanism for distributing the cryptographic keys to authorized parties. A small (on the order of several hundred bytes), fixed-size per file, *trusted storage* is available for authentication data.

We assume that the storage servers are actively controlled by an adversary. The adversary can adaptively

alter the data stored on the storage servers or perform any other attack on the stored data, but it cannot modify or observe the trusted storage. A particularly interesting attack that the adversary can mount is a *replay attack*, in which stale data is returned to read requests of clients. Using the trusted storage to keep some constant-size information per file, and keeping more information per file on untrusted storage, our goal is to design and evaluate integrity algorithms that allow the update and verification of individual blocks in files and that detect data modification and replay attacks.

In our framework, a file  $F$  is divided into  $n$  fixed-size blocks  $B_1 B_2 \dots B_n$  (the last block  $B_n$  might be shorter than the first  $n - 1$  blocks), each encrypted individually with the encryption key of the file and stored on the untrusted storage servers ( $n$  differs per file). The constant-size, trusted storage for file  $F$  is denoted  $TS_F$ . Additional storage for file  $F$ , which can reside in untrusted storage, is denoted  $US_F$ ; of course,  $US_F$  can be written to the untrusted storage server.

The storage interface provides two basic operations to the clients:  $F.\text{WriteBlock}(i, C)$  stores content  $C$  at block index  $i$  in file  $F$  and  $C \leftarrow F.\text{ReadBlock}(i)$  reads (encrypted) content from block index  $i$  in file  $F$ . An integrity algorithm for an encrypted file system consists of five operations. In the initialization algorithm  $\text{Init}$  for file  $F$ , the encryption key for the file is generated. In an update operation  $\text{Update}(i, B)$  for file  $F$ , an authorized client updates the  $i$ -th block in the file with the encryption of block content  $B$  and updates the integrity information for the  $i$ -th block stored in  $TS_F$  and  $US_F$ . In the check operation  $\text{Check}(i, C)$  for file  $F$ , an authorized client first decrypts  $C$  and then checks that the decrypted block content is authentic, using the additional storage  $TS_F$  and  $US_F$  for file  $F$ . The check operation returns the decrypted block if it concludes that the block content is authentic and  $\perp$  otherwise. A client can additionally perform an append operation  $\text{Append}(B)$  for file  $F$ , in which a new block that contains the encryption of  $B$  is appended to the encrypted file, and a Delete operation that deletes the last block in a file and updates the integrity information for the file.

Using the algorithms we have defined for an integrity scheme for an encrypted file, a client can read or write at any byte offset in the file. For example, to write to a byte offset that is not at a block boundary, the client first reads the block to which the byte offset belongs, decrypts it and checks its integrity using algorithm  $\text{Check}$ . Then, the client constructs the new data block by replacing the appropriate bytes in the decrypted block, and calls  $\text{Update}$  to encrypt the new block and compute its integrity information.

In designing an integrity algorithm for a cryptographic file system, we consider the following metrics. First is the size of the untrusted storage  $US_F$ ; we will always enforce that the trusted storage  $TS_F$  is of constant size, independent of the number of blocks. Second is the integrity bandwidth for updating and checking individual file blocks, defined as the number of bytes from  $US_F$  accessed (updated or read) when accessing a block of file  $F$ , averaged over either: all blocks in  $F$  when we speak of a per-file integrity bandwidth; all blocks in all files when we speak of the integrity bandwidth of the file system; or all blocks accessed in a particular trace when we speak of one trace. Third is the performance cost of writing and reading files.

### 3 Preliminaries

#### 3.1 Merkle Trees

Merkle trees [24] are used to authenticate  $n$  data items with constant-size trusted storage. A Merkle tree for data items  $M_1, \dots, M_n$ , denoted  $MT(M_1, \dots, M_n)$ , is a binary tree that has  $M_1, \dots, M_n$  as leaves. An interior node of the tree with children  $C_L$  and  $C_R$  is the hash of the concatenation of its children (i.e.,  $h(C_L || C_R)$ , for  $h : \{0, 1\}^* \rightarrow \{0, 1\}^s$  a second preimage resistant hash function [29] that outputs strings of length  $s$  bits). If the root of the tree is stored in trusted storage, then all the leaves of the tree can be authenticated by reading from the tree a number of hashes logarithmic in  $n$ .

We define the Merkle tree for a file  $F$  with  $n$  blocks  $B_1, \dots, B_n$  to be the binary tree  $MT_F = MT(h(1 || B_1), \dots, h(n || B_n))$ . A Merkle tree with a given set of leaves can be constructed in multiple ways. We choose to append a new block in the tree as a right-most child, so that the tree has the property that all the left subtrees are complete. We define several algorithms for a Merkle tree  $T$ , for which we omit the implementation details, due to space limitations.

- In the  $UpdateTree(R, i, hval)$  algorithm for tree  $T$ , the hash stored at the  $i$ -th leaf of  $T$  (counting from left to right) is updated to  $hval$ . This triggers an update of all the hashes stored on the path from the  $i$ -th leaf to the root of the tree. It is necessary to first check that all the siblings of the nodes on the path from the updated leaf to the root of the tree are authentic. Finally, the updated root of the tree is output in  $R$ .

- The  $CheckTree(R, i, hval)$  algorithm for tree  $T$  checks that the hash stored at the  $i$ -th leaf matches  $hval$ . All the hashes stored at the nodes on the path from the  $i$ -th leaf to the root are computed and the root of  $T$  is checked finally to match the value stored in  $R$ .

- Algorithm  $AppendTree(R, hval)$  for tree  $T$  appends a new leaf  $u$  that stores the hash value  $hval$  to the tree, updates the path from this new leaf to the root of the tree and outputs the new root of the tree in  $R$ .

- The  $DeleteTree(R)$  algorithm for tree  $T$  deletes the last leaf from the tree, updates the remaining path to the root of the tree and outputs the new root of the tree in  $R$ .

#### 3.2 Encryption Schemes and Tweakable Ciphers

An encryption scheme consists of a key generation algorithm  $Gen$  that outputs an encryption key, an encryption algorithm  $E_k(M)$  that outputs the encryption of a message  $M$  with secret key  $k$  and a decryption algorithm  $D_k(C)$  that outputs the decryption of a ciphertext  $C$  with secret key  $k$ . A widely used secure encryption scheme is AES [2] in CBC mode [8].

A *tweakable cipher* [21, 15] is, informally, a length-preserving encryption method that uses a *tweak* in both the encryption and decryption algorithms for variability. A tweakable encryption of a message  $M$  with tweak  $t$  and secret key  $k$  is denoted  $E_k^t(M)$  and, similarly, the decryption of ciphertext  $C$  with tweak  $t$  and secret key  $k$  is denoted  $D_k^t(C)$ . The tweak is a public parameter, and the security of the tweakable cipher is based only on the secrecy of the encryption key. Tweakable ciphers can be used to encrypt fixed-size blocks written to disk in a file system. Suitable values of the tweak for this case are, for example, block addresses or block indices in the file. There is a distinction between *narrow-block tweakable ciphers* that operate on block lengths of 128 bits (as regular block ciphers) and *wide-block tweakable ciphers* that operate on arbitrarily large blocks (e.g., 512 bytes or 4KB). In this paper we use the term tweakable ciphers to refer to wide-block tweakable ciphers as defined by Halevi and Rogaway [15].

The security of tweakable ciphers implies an interesting property, called *non-malleability* [15], that guarantees that if only a single bit is changed in a valid ciphertext, then its decryption is indistinguishable from a random plaintext. Tweakable cipher constructions include CMC [15] and EME [16].

#### 3.3 Efficient Block Integrity Using Randomness of Block Contents

Oprea et al. [26] provide an efficient integrity construction in a block-level storage system. This integrity construction is based on the experimental observation that contents of blocks written to disk usually are efficiently

distinguishable from *random blocks*, i.e., blocks uniformly chosen at random from the set of all blocks of a fixed length. Assuming that data blocks are encrypted with a tweakable cipher, the integrity of the blocks that are efficiently distinguishable from random blocks can be checked by performing a randomness test on the block contents. The non-malleability property of tweakable ciphers implies that if block contents after decryption are distinguishable from random, then it is very likely that the contents are authentic. This idea permits a reduction in the trusted storage needed for checking block integrity: a hash is stored only for those (usually few) blocks that are indistinguishable from random blocks (or, in short, *random-looking blocks*).

An example of a statistical test `IsRand` [26] that can be used to distinguish block contents from random blocks evaluates the entropy of a block and considers random those blocks that have an entropy higher than a threshold chosen experimentally. For a block  $B$ , `IsRand( $B$ )` returns 1 with high probability if  $B$  is a uniformly random block in the block space and 0, otherwise. Oprea et al. [26] provide an upper bound on the false negative rate of the randomness test that is used in the security analysis of the scheme.

We use the ideas from Oprea et al. [26] as a starting point for our first algorithm for implementing file integrity in cryptographic file systems. The main challenge to construct integrity algorithms in our model is to efficiently reduce the amount of trusted storage per file to a constant value. Our second algorithm also exploits the redundancy in file contents to reduce the additional space for integrity, but in a different way, by embedding a message authentication code (MAC) in file blocks that can be compressed enough. Both of these schemes build from a novel technique that is described in Section 4.1 for efficiently tracking the number of writes to file blocks.

## 4 Write Counters for File Blocks

All the integrity constructions for encrypted storage described in the next section use write counters for the blocks in a file. A write counter for a block denotes the total number of writes done to that block index. Counters are used to reduce the additional storage space taken by encrypting with a block cipher in CBC mode, as described in Section 4.2. Counters are also a means of distinguishing different writes performed to the same block address and as such, can be used to prevent against replay attacks.

We define several operations for the write counters of the blocks in a file  $F$ . The `UpdateCtr( $i$ )` algorithm either initializes the value of the counter for the  $i$ -th block

in file  $F$  with 1, or it increments the counter for the  $i$ -th block if it has already been initialized. The algorithm also updates the information for the counters stored in  $US_F$ . Function `GetCtr( $i$ )` returns the value of the counter for the  $i$ -th block in file  $F$ . When counters are used to protect against replay attacks, they need to be authenticated with a small amount of trusted storage. For authenticating block write counters, we define an algorithm `AuthCtr` that modifies the trusted storage space  $TS_F$  of file  $F$  to contain the trusted authentication information for the write counters of  $F$ , and a function `CheckCtr` that checks the authenticity of the counters stored in  $US_F$  using the trusted storage  $TS_F$  for file  $F$  and returns true if the counters are authentic and false, otherwise. Both operations for authenticating counters are invoked by an authorized client.

### 4.1 Storage and Authentication of Block Write Counters

A problem that needs to be addressed in the design of the various integrity algorithms described below is the storage and authentication of the block write counters. If a counter per file block were used, this would result in significant additional storage for counters. Here we propose a more efficient method of storing the block write counters, based on analyzing the file access patterns in NFS traces collected at Harvard University [9].

**Counter intervals.** We performed experiments on the NFS Harvard traces [9] in order to analyze the file access patterns. We considered three different traces (LAIR, DEASNA and HOME02) for a period of one week. The LAIR trace consists of research workload traces from Harvard's computer science department. The DEASNA trace is a mix of research and email workloads from the division of engineering and applied sciences at Harvard. HOME02 is mostly the email workload from the campus general purpose servers.

Ellard et al. [9] make the observation that a large number of file accesses are sequential. This leads to the idea that the values of the write counters for adjacent blocks in a file might be correlated. To test this hypothesis, we represent counters for blocks in a file using *counter intervals*. A counter interval is defined as a sequence of consecutive blocks in a file that all share the same value of the write counter. For a counter interval, we need to store only the beginning and end of the interval, and the value of the write counter.

Table 1 shows the average storage per file used by the two counter representation methods for the three traces. We represent a counter using 2 bytes (as the maximum

observed value of a counter was 9905) and we represent file block indices with 4 bytes. The counter interval method reduces the average storage needed for counters by a factor of 30 for the LAIR trace, 26.5 for the DEASNA trace and 7.66 for the HOME02 trace compared to the method that stores a counter per file block. This justifies our design choice to use counter intervals for representing counter values in the integrity algorithms presented in the next section.

	LAIR	DEASNA	HOME02
Counter per block	547.8 bytes	1.46 KB	3.16 KB
Counter intervals	18.35 bytes	55.04 bytes	413.44 bytes

Table 1: Average storage per file for two counter representation methods.

**Counter representation.** The counter intervals for file  $F$  are represented by two arrays:  $\text{IntStart}_F$  keeps the block indices where new counter intervals start and  $\text{CtrVal}_F$  keeps the values of the write counter for each interval. The trusted storage  $\text{TS}_F$  for file  $F$  includes either the arrays  $\text{IntStart}_F$  and  $\text{CtrVal}_F$  if they fit into  $\text{TS}_F$  or, for each array, a hash of all its elements (concatenated), otherwise. In the limit, to reduce the bandwidth for integrity, we could build a Merkle tree to authenticate each of these arrays and store the root of these trees in  $\text{TS}_F$ , but we have not seen in the Harvard traces files that would warrant this. We omit here the implementation details for the `UpdateCtr`, `GetCtr`, `AuthCtr` and `CheckCtr` operations on counters, due to space limitations.

If the counter intervals for a file get too dispersed, then the size of the arrays  $\text{IntStart}_F$  and  $\text{CtrVal}_F$  might increase significantly. To keep the untrusted storage for integrity low, we could periodically change the encryption key for the file, re-encrypt all blocks in the file, and reset the block write counters to 0.

## 4.2 Length-Preserving Stateful Encryption with Counters

Secure encryption schemes are usually not length-preserving. However, one of our design goals stated in the introduction is to add security (and, in particular, encryption) to file systems in a manner transparent to the storage servers. For this purpose, we introduce here the notion of a *length-preserving stateful encryption scheme* for a file  $F$ , an encryption scheme that encrypts blocks in a way that preserves the length of the original blocks, and stores any additional information in the untrusted storage space for the file. We define a length-preserving stateful encryption scheme for a file  $F$  to consist of a key generation algorithm  $G^{\text{len}}$  that generates an encryption key for the file, an encryption algorithm  $E^{\text{len}}$  that encrypts

block content  $B$  for block index  $i$  with key  $k$  and outputs ciphertext  $C$ , and a decryption algorithm  $D^{\text{len}}$  that decrypts the encrypted content  $C$  of block  $i$  with key  $k$  and outputs the plaintext  $B$ . Both the  $E^{\text{len}}$  and  $D^{\text{len}}$  algorithms also modify the untrusted storage space for the file.

Tweakable ciphers are by definition length-preserving stateful encryption schemes. A different construction on which we elaborate below uses write counters for file blocks. Let  $(\text{Gen}, E, D)$  be an encryption scheme constructed from a block cipher in CBC mode. To encrypt an  $n$ -block message in the CBC encryption mode, a random initialization vector is chosen. The ciphertext consists of  $n + 1$  blocks, with the first being the initialization vector. We denote by  $E_k(B, iv)$  the output of the encryption of  $B$  (excluding the initialization vector) using key  $k$  and initialization vector  $iv$ , and similarly by  $D_k(C, iv)$  the decryption of  $C$  using key  $k$  and initialization vector  $iv$ .

We replace the random initialization vectors for encrypting a file block with a pseudorandom function application of the block index concatenated with the write counter for the block. This is intuitively secure because different initialization vectors are used for different encryptions of the same block, and moreover, the properties of pseudorandom functions imply that the initialization vectors are indistinguishable from random. It is thus enough to store the write counters for the blocks of a file, and the initialization vectors for the file blocks can be easily inferred.

The  $G^{\text{len}}$ ,  $E^{\text{len}}$  and  $D^{\text{len}}$  algorithms for a file  $F$  are described in Figure 1. Here  $\text{PRF} : \mathcal{K}_{\text{PRF}} \times \mathcal{I} \rightarrow \mathcal{B}$  denotes a pseudorandom function family with key space  $\mathcal{K}_{\text{PRF}}$ , input space  $\mathcal{I}$  (i.e., the set of all block indices concatenated with block counter values), and output space  $\mathcal{B}$  (i.e., the block space of  $E$ ).

## 5 Integrity Constructions for Encrypted Storage

In this section, we first present a Merkle tree integrity construction for encrypted storage, used in file systems such as Cepheus [10], FARSITE [1], and Plutus [17]. Second, we introduce a new integrity construction based on tweakable ciphers that uses some ideas from Oprea et al. [26]. Third, we give a new construction based on compression levels of block contents. We evaluate the performance of the integrity algorithms described here in Section 7.

$G^{\text{len}}(F)$ : $k_1 \xleftarrow{R} \mathcal{K}_{\text{PRF}}$ $k_2 \leftarrow \text{Gen}()$ return $\langle k_1, k_2 \rangle$	$E_{\langle k_1, k_2 \rangle}^{\text{len}}(F, i, B)$ : $F.\text{UpdateCtr}(i)$ $iv \leftarrow \text{PRF}_{k_1}(i    F.\text{GetCtr}(i))$ $C \leftarrow E_{k_2}(B, iv)$ return $C$	$D_{\langle k_1, k_2 \rangle}^{\text{len}}(F, i, C)$ : $iv \leftarrow \text{PRF}_{k_1}(i    F.\text{GetCtr}(i))$ $B \leftarrow D_{k_2}(C, iv)$ return $B$
--	---	--

Figure 1: Implementing a length-preserving stateful encryption scheme with write counters.

$F.\text{Update}(i, B)$ : $k \leftarrow F.\text{enc\_key}$ $\text{MT}_F.\text{UpdateTree}(\text{TS}_F, i, h(i    B))$ $C \leftarrow E_k^{\text{len}}(F, i, B)$ $F.\text{WriteBlock}(i, C)$	$F.\text{Check}(i, C)$ : $k \leftarrow F.\text{enc\_key}$ $B_i \leftarrow D_k^{\text{len}}(F, i, C)$ if $\text{MT}_F.\text{CheckTree}(\text{TS}_F, i, h(i    B_i)) = \text{true}$ return $B_i$ else return $\perp$
$F.\text{Append}(B)$ : $k \leftarrow F.\text{enc\_key}$ $n \leftarrow F.\text{blocks}$ $\text{MT}_F.\text{AppendTree}(\text{TS}_F, h(n + 1    B))$ $C \leftarrow E_k^{\text{len}}(F, n + 1, B)$ $F.\text{WriteBlock}(n + 1, C)$	$F.\text{Delete}()$ : $n \leftarrow F.\text{blocks}$ $\text{MT}_F.\text{DeleteTree}(\text{TS}_F)$ delete $B_n$ from file $F$

Figure 2: The Update, Check, Append and Delete algorithms for the MT-EINT construction.

## 5.1 The Merkle Tree Construction MT-EINT

In this construction, file blocks can be encrypted with any length-preserving stateful encryption scheme and they are authenticated with a Merkle tree. More precisely, if  $F$  is a file comprised of blocks  $B_1, \dots, B_n$ , then the untrusted storage for integrity for file  $F$  is  $\text{US}_F = \text{MT}_F(h(1 || B_1), \dots, h(n || B_n))$  (for  $h$  a second-preimage resistant hash function), and the trusted storage  $\text{TS}_F$  is the root of this tree.

The algorithm Init runs the key generation algorithm  $G^{\text{len}}$  of the length-preserving stateful encryption scheme for file  $F$ . The algorithms Update, Check, Append and Delete of the MT-EINT construction are given in Figure 2. We denote here by  $F.\text{enc\_key}$  the encryption key for file  $F$  (generated in the Init algorithm) and  $F.\text{blocks}$  the number of blocks in file  $F$ .

- In the  $\text{Update}(i, B)$  algorithm for file  $F$ , the  $i$ -th leaf in  $\text{MT}_F$  is updated with the hash of the new block content using the algorithm  $\text{UpdateTree}$  and the encryption of  $B$  is stored in the  $i$ -th block of  $F$ .

- To append a new block  $B$  to file  $F$  with algorithm  $\text{Append}(B)$ , a new leaf is appended to  $\text{MT}_F$  with the algorithm  $\text{AppendTree}$ , and then an encryption of  $B$  is stored in the  $(n + 1)$ -th block of  $F$  (for  $n$  the number of blocks of  $F$ ).

- In the  $\text{Check}(i, C)$  algorithm for file  $F$ , block  $C$  is decrypted, and its integrity is checked using the  $\text{CheckTree}$  algorithm.

- To delete the last block from a file  $F$  with algorithm  $\text{Delete}$ , the last leaf in  $\text{MT}_F$  is deleted with the algorithm  $\text{DeleteTree}$ .

The MT-EINT construction detects data modification

and block swapping attacks, as file block contents are authenticated by the root of the Merkle tree for each file. The MT-EINT construction is also secure against replay attacks, as the tree contains the hashes of the latest version of the data blocks and the root of the Merkle tree is authenticated in trusted storage.

## 5.2 The Randomness Test Construction RAND-EINT

Whereas in the Merkle tree construction any length-preserving stateful encryption algorithm can be used to individually encrypt blocks in a file, the randomness test construction uses the observation from Oprea et al. [26] that the integrity of the blocks that are efficiently distinguishable from random blocks can be checked with a randomness test if a tweakable cipher is used to encrypt them. As such, integrity information is stored only for random-looking blocks.

In this construction, a Merkle tree per file that authenticates the contents of the random-looking blocks is built. The untrusted storage for integrity  $\text{US}_F$  for file  $F$  comprised of blocks  $B_1, \dots, B_n$  includes this tree  $\text{RTree}_F = \text{MT}(h(i || B_i) : i \in \{1, \dots, n\} \text{ and } \text{IsRand}(B_i) = 1)$ , and, in addition, the set of block numbers that are random-looking  $\text{RArr}_F = \{i \in \{1, \dots, n\} : \text{IsRand}(B_i) = 1\}$ , ordered the same as the leaves in the previous tree  $\text{RTree}_F$ . The root of the tree  $\text{RTree}_F$  is kept in the trusted storage  $\text{TS}_F$  for file  $F$ .

To prevent against replay attacks, clients need to distinguish different writes of the same block in a file. A simple idea [26] is to use a counter per file block that denotes the number of writes of that block, and make the

<pre> <i>F.Update</i>(<i>i, B</i>) :   <i>k</i> ← <i>F.enc_key</i>   <i>F.UpdateCtr</i>(<i>i</i>)   <i>F.AuthCtr</i>()   if <i>IsRand</i>(<i>B</i>) = 0     if <i>i</i> ∈ <i>RArr<sub>F</sub></i>       <i>RTree<sub>F</sub></i>.<i>DelOffsetTree</i>(<i>TS<sub>F</sub></i>, <i>RArr<sub>F</sub></i>, <i>i</i>)     else       if <i>i</i> ∈ <i>RArr<sub>F</sub></i>         <i>j</i> ← <i>RArr<sub>F</sub></i>.<i>SearchOffset</i>(<i>i</i>)         <i>RTree<sub>F</sub></i>.<i>UpdateTree</i>(<i>TS<sub>F</sub></i>, <i>j</i>, <i>h(i  B)</i>)       else         <i>RTree<sub>F</sub></i>.<i>AppendTree</i>(<i>TS<sub>F</sub></i>, <i>h(i  B)</i>)         append <i>i</i> at end of <i>RArr<sub>F</sub></i>   <i>F.WriteBlock</i>(<i>i, E<sub>k</sub><sup>F.Tweak</sup>(i, F.GetCtr(i)) (B)</i>) </pre>	<pre> <i>F.Check</i>(<i>i, C</i>):   <i>k</i> ← <i>F.enc_key</i>   if <i>F.CheckCtr</i>() = false     return ⊥   <i>B<sub>i</sub></i> ← <i>D<sub>k</sub><sup>F.Tweak</sup>(i, F.GetCtr(i)) (C)</i>   if <i>IsRand</i>(<i>B<sub>i</sub></i>) = 0     return <i>B<sub>i</sub></i>   else     if <i>i</i> ∈ <i>RArr<sub>F</sub></i>       <i>j</i> ← <i>RArr<sub>F</sub></i>.<i>SearchOffset</i>(<i>i</i>)       if <i>RTree<sub>F</sub></i>.<i>CheckTree</i>(<i>TS<sub>F</sub></i>, <i>j</i>, <i>h(i  B<sub>i</sub>)</i>) = true         return <i>B<sub>i</sub></i>       else         return ⊥     else       return ⊥ </pre>
<pre> <i>F.Append</i>(<i>B</i>):   <i>k</i> ← <i>F.enc_key</i>   <i>n</i> ← <i>F.blocks</i>   <i>F.UpdateCtr</i>(<i>n + 1</i>)   <i>F.AuthCtr</i>()   if <i>IsRand</i>(<i>B</i>) = 1     <i>RTree<sub>F</sub></i>.<i>AppendTree</i>(<i>TS<sub>F</sub></i>, <i>h(n + 1  B)</i>)     append <i>n + 1</i> at end of <i>RArr<sub>F</sub></i>   <i>F.WriteBlock</i>(<i>n + 1, E<sub>k</sub><sup>F.Tweak</sup>(n+1, F.GetCtr(n+1)) (B)</i>) </pre>	<pre> <i>F.Delete</i>():   <i>n</i> ← <i>F.blocks</i>   if <i>n</i> ∈ <i>RArr<sub>F</sub></i>     <i>RTree<sub>F</sub></i>.<i>DelOffsetTree</i>(<i>TS<sub>F</sub></i>, <i>RArr<sub>F</sub></i>, <i>n</i>)   delete <i>B<sub>n</sub></i> from file <i>F</i> </pre>

Figure 3: The Update, Check, Append and Delete algorithms for the RAND-EINT construction.

counter part of the encryption tweak. The block write counters need to be authenticated in the trusted storage space for the file  $F$  to prevent clients from accepting valid older versions of a block that are considered not random by the randomness test. To ensure that file blocks are encrypted with different tweaks, we define the tweak for a file block to be a function of the file, the block index and the block write counter. We denote by  $F.Tweak$  the tweak-generating function for file  $F$  that takes as input a block index and a block counter and outputs the tweak for that file block. The properties of tweakable ciphers imply that if a block is decrypted with a different counter (and so a different tweak), then it will look random with high probability.

The algorithm  $Init$  selects a key at random from the key space of the tweakable encryption scheme  $E$ . The Update, Check, Append and Delete algorithms of RAND-EINT are detailed in Figure 3. For the array  $RArr_F$ ,  $RArr_F.items$  denotes the number of items in the array,  $RArr_F.last$  denotes the last element in the array, and the function  $SearchOffset(i)$  for the array  $RArr_F$  gives the position in the array where index  $i$  is stored (if it exists in the array).

- In the  $Update(i, B)$  algorithm for file  $F$ , the write counter for block  $i$  is incremented and the counter authentication information from  $TS_F$  is updated with the algorithm  $AuthCtr$ . Then, the randomness test  $IsRand$  is applied to block content  $B$ . If  $B$  is not random looking, then the leaf corresponding to block  $i$  (if it exists) has to be removed from  $RTree_F$ . This is done with the algorithm  $DelOffsetTree$ , described in Figure 4. On the

other hand, if  $B$  is random-looking, then the leaf corresponding to block  $i$  has to be either updated with the new hash (if it exists in the tree) or appended in  $RTree_F$ . Finally, the tweakable encryption of  $B$  is stored in the  $i$ -th block of  $F$ .

- To append a new block  $B$  to file  $F$  with  $n$  blocks using the  $Append(B)$  algorithm, the counter for block  $n + 1$  is updated first with algorithm  $UpdateCtr$ . The counter authentication information from trusted storage is also updated with algorithm  $AuthCtr$ . Furthermore, the hash of the block index concatenated with the block content is added to  $RTree_F$  only if the block is random-looking. In addition, index  $n + 1$  is added to  $RArr_F$  in this case. Finally, the tweakable encryption of  $B$  is stored in the  $(n + 1)$ -th block of  $F$ .

- In the  $Check(i, C)$  algorithm for file  $F$ , the authentication information for the block counters is checked first. Then block  $C$  is decrypted, and checked for integrity. If the content of the  $i$ -th block is not random-looking, then by the properties of tweakable ciphers we can infer that the block is valid with high probability. Otherwise, the integrity of the  $i$ -th block is checked using the tree  $RTree_F$ . If  $i$  is not a block index in the tree, then the integrity of block  $i$  is unconfirmed and the block is rejected.

- In the Delete algorithm for file  $F$ , the hash of the last block has to be removed from the tree by calling the algorithm  $DelOffsetTree$  (described in Figure 4), in the case in which the last block is authenticated through  $RTree_F$ .

It is not necessary to authenticate in trusted storage the array  $RArr_F$  of indices of the random-looking blocks in

```

T.DelOffsetTree(TSF, RArrF, i):
  j ← RArrF.SearchOffset(i)
  l ← RArrF.last
  if j ≠ l
    T.UpdateTree(TSF, j, h(l||Bi))
    RArrF[j] ← l
    RArrF.items ← RArrF.items - 1
  T.DeleteTree(TSF)

```

Figure 4: The DelOffsetTree algorithm for a tree  $T$  deletes the hash of block  $i$  from  $T$  and moves the last leaf to its position, if necessary.

a file. The reason is that the root of  $RTree_F$  is authenticated in trusted storage and this implies that an adversary cannot modify the order of the leaves in  $RTree_F$  without being detected in the AppendTree, UpdateTree or CheckTree algorithms.

The construction RAND-EINT protects against unauthorized modification of data written to disk and block swapping attacks by authenticating the root of  $RTree_F$  in the trusted storage space for each file. By using write counters in the encryption of block contents and authenticating the values of the counters in trusted storage, this construction provides defense against replay attacks and provides all the security properties of the MT-EINT construction.

### 5.3 The Compression Construction COMP-EINT

This construction is again based on the intuition that many workloads feature redundancy in file contents. In this construction, the block is compressed before encryption. If the compression level of the block content is high enough, then a message authentication code (i.e., MAC) of the block can be stored in the block itself, reducing the amount of storage necessary for integrity. The authentication information for blocks that can be compressed is stored on untrusted storage, and consequently a MAC is required. Like in the previous construction, a Merkle tree  $RTree_F$  is built over the hashes of the blocks in file  $F$  that cannot be compressed enough, and the root of the tree is kept in trusted storage. In order to prevent replay attacks, it is necessary that block write counters are included either in the computation of the block MAC (in the case in which the block can be compressed) or in hashing the block (in the case in which the block cannot be compressed enough). Similarly to scheme RAND-EINT, the write counters for a file  $F$  need to be authenticated in the trusted storage space  $TS_F$ .

In this construction, file blocks can be encrypted with any length-preserving encryption scheme, as defined in Section 4.2. In describing the scheme, we need

compression and decompression algorithms such that  $\text{decompress}(\text{compress}(m)) = m$ , for any message  $m$ . We can also pad messages up to a certain fixed length by using the pad function with an output of  $l$  bytes, and unpad a padded message with the unpad function such that  $\text{unpad}(\text{pad}(m)) = m$ , for all messages  $m$  of length less than  $l$  bytes. We can use standard padding methods for implementing these algorithms [4]. To authenticate blocks that can be compressed, we use a message authentication code  $H : \mathcal{K}_H \times \{0, 1\}^* \rightarrow \{0, 1\}^s$  that outputs strings of length  $s$  bits.

The algorithm Init runs the key generation algorithm  $G^{\text{len}}$  of the length-preserving stateful encryption scheme for file  $F$  to generate key  $k_1$  and selects at random a key  $k_2$  from the key space  $\mathcal{K}_H$  of  $H$ . It outputs the tuple  $\langle k_1, k_2 \rangle$ . The Update, Append, Check and Delete algorithms of the COMP-EINT construction are detailed in Figure 5. Here  $L_c$  is the byte length of the largest plaintext size for which the ciphertext is of length at most the file block length less the size of a MAC function output. For example, if the block size is 4096 bytes, HMAC [3] with SHA-1 is used for computing MACs (whose output is 20 bytes) and 16-byte AES is used for encryption, then  $L_c$  is the largest multiple of the AES block size (i.e., 16 bytes) less than  $4096 - 20 = 4076$  bytes. The value of  $L_c$  in this case is 4064 bytes.

- In the Update( $i, B$ ) algorithm for file  $F$ , the write counter for block  $i$  is incremented and the counter authentication information from  $TS_F$  is updated with the algorithm AuthCtr. Then block content  $B$  is compressed to  $B^c$ . If the length of  $B^c$  (denoted  $|B^c|$ ) is at most  $L_c$ , then there is room to store the MAC of the block content inside the block. In this case, the hash of the previous block content stored at the same address is deleted from the Merkle tree  $RTree_F$ , if necessary. The compressed block is padded and encrypted, and then stored with its MAC in the  $i$ -th block of  $F$ . Otherwise, if the block cannot be compressed enough, then its hash has to be inserted into the Merkle tree  $RTree_F$ . The block content  $B$  is then encrypted with a length-preserving stateful encryption scheme using the key for the file and is stored in the  $i$ -th block of  $F$ .

- To append a new block  $B$  to file  $F$  with  $n$  blocks using the Append( $B$ ) algorithm, the counter for block  $n+1$  is updated first with algorithm UpdateCtr. The counter authentication information from trusted storage is also updated with algorithm AuthCtr. Block  $B$  is then compressed. If it has an adequate compression level, then the compressed block is padded and encrypted, and a MAC is concatenated at the end of the new block. Otherwise, a new hash is appended to the Merkle tree  $RTree_F$  and an encryption of  $B$  is stored in the  $(n+1)$ -th block of  $F$ .

<pre> F.Update(<math>i, B</math>) : <math>\langle k_1, k_2 \rangle \leftarrow F.\text{enc\_key}</math> F.UpdateCtr(<math>i</math>) F.AuthCtr() <math>B^c \leftarrow \text{compress}(B)</math> if <math> B^c  \leq L_c</math>   if <math>i \in \text{RArr}_F</math>     RTree<math>_F</math>.DelOffsetTree(TS<math>_F</math>, RArr<math>_F</math>, <math>i</math>)     <math>C \leftarrow E_{k_1}^{\text{len}}(F, i, \text{pad}(B^c))</math>     F.WriteBlock(<math>i, C    H_{k_2}(i    F.\text{GetCtr}(i)    B)</math>)   else     if <math>i \in \text{RArr}_F</math>       <math>j \leftarrow \text{RArr}_F.\text{SearchOffset}(i)</math>       RTree<math>_F</math>.UpdateTree(TS<math>_F</math>, <math>j</math>, <math>h(i    F.\text{GetCtr}(i)    B)</math>)     else       RTree<math>_F</math>.AppendTree(TS<math>_F</math>, <math>h(i    F.\text{GetCtr}(i)    B)</math>)       append <math>i</math> at end of RArr<math>_F</math>     <math>C \leftarrow E_{k_1}^{\text{len}}(F, i, B)</math>     F.WriteBlock(<math>i, C</math>) </pre>	<pre> F.Check(<math>i, C</math>): <math>\langle k_1, k_2 \rangle \leftarrow F.\text{enc\_key}</math> if F.CheckCtr() = false   return <math>\perp</math> if <math>i \in \text{RArr}_F</math>   <math>B_i \leftarrow D_{k_1}^{\text{len}}(F, i, C)</math>   <math>j \leftarrow \text{RArr}_F.\text{SearchOffset}(i)</math>   if RTree<math>_F</math>.CheckTree(TS<math>_F</math>, <math>j</math>,     <math>h(i    F.\text{GetCtr}(i)    B_i)</math>) = true     return <math>B_i</math>   else     return <math>\perp</math> else   parse <math>C</math> as <math>C'    hval</math>   <math>B_i^c \leftarrow \text{unpad}(D_{k_1}^{\text{len}}(F, i, C'))</math>   <math>B_i \leftarrow \text{decompress}(B_i^c)</math>   if <math>hval = H_{k_2}(i    F.\text{GetCtr}(i)    B_i)</math>     return <math>B_i</math>   else     return <math>\perp</math> </pre>
<pre> F.Append(<math>B</math>) : <math>\langle k_1, k_2 \rangle \leftarrow F.\text{enc\_key}</math> <math>n \leftarrow F.\text{blocks}</math> F.UpdateCtr(<math>n + 1</math>) F.AuthCtr() <math>B^c \leftarrow \text{compress}(B)</math> if <math> B^c  \leq L_c</math>   <math>C \leftarrow E_{k_1}^{\text{len}}(F, n + 1, \text{pad}(B^c))</math>   F.WriteBlock(<math>i, C    H_{k_2}(n + 1    F.\text{GetCtr}(n + 1)    B)</math>) else   RTree<math>_F</math>.AppendTree(TS<math>_F</math>, <math>h(n + 1    F.\text{GetCtr}(n + 1)    B)</math>)   append <math>n + 1</math> at end of RArr<math>_F</math>   <math>C \leftarrow E_{k_1}^{\text{len}}(F, n + 1, B)</math>   F.WriteBlock(<math>n + 1, C</math>) </pre>	<pre> F.Delete(): <math>n \leftarrow F.\text{blocks}</math> if <math>n \in \text{RArr}_F</math>   RTree<math>_F</math>.DelOffsetTree(TS<math>_F</math>, RArr<math>_F</math>, <math>n</math>)   delete <math>B_n</math> from file <math>F</math> </pre>

Figure 5: The Update, Check, Append and Delete algorithms for the COMP-EINT construction.

- In the Check( $i, C$ ) algorithm for file  $F$ , the authentication information from TS $_F$  for the block counters is checked first. There are two cases to consider. First, if the  $i$ -th block of  $F$  is authenticated through the Merkle tree RTree $_F$ , as indicated by RArr $_F$ , then the block is decrypted and algorithm CheckTree is called. Otherwise, the MAC of the block content is stored at the end of the block and we can thus parse the  $i$ -th block of  $F$  as  $C' || hval$ .  $C'$  has to be decrypted, unpadded and decompressed, in order to obtain the original block content  $B_i$ . The value  $hval$  stored in the block is checked to match the MAC of the block index  $i$  concatenated with the write counter for block  $i$  and block content  $B_i$ .

- In the Delete algorithm for file  $F$ , the hash of the last block has to be removed from the tree by calling the algorithm DelOffsetTree (described in Figure 4), in the case in which the last block is authenticated through RTree $_F$ .

The construction COMP-EINT prevents against replay attacks by using write counters for either computing a MAC over the contents of blocks that can be compressed enough, or a hash over the contents of blocks that cannot be compressed enough, and authenticating the write counters in trusted storage. It meets all the security properties of MT-EINT and RAND-EINT.

## 6 Implementation

Our integrity algorithms are very general and they can be integrated into any cryptographic file system in either the kernel or user space. For the purpose of evaluating and comparing their performance, we implemented them in EncFS [14], an open-source user-level file system that transparently encrypts file blocks. EncFS uses the FUSE [12] library to provide the file system interface. FUSE provides a simple library API for implementing file systems and it has been integrated into recent versions of the Linux kernel.

In EncFS, files are divided into fixed-size blocks and each block is encrypted individually. Several ciphers such as AES and Blowfish in CBC mode are available for block encryption. We implemented in EncFS the three constructions that provide integrity: MT-EINT, RAND-EINT and COMP-EINT. While any length-preserving encryption scheme can be used in the MT-EINT and COMP-EINT constructions, RAND-EINT is constrained to use a tweakable cipher for encrypting file blocks. We choose to encrypt file blocks in MT-EINT and COMP-EINT with the length-preserving stateful encryption derived from the AES cipher in CBC mode (as shown in Section 4.2), and use the

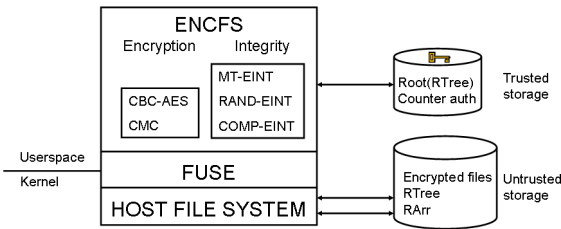


Figure 6: Prototype architecture.

CMC tweakable cipher [15] as the encryption method in RAND-EINT. In our integrity algorithms, we use the SHA-1 hash function and the message-authentication code HMAC instantiated also with the SHA-1 hash function. For compressing and decompressing blocks in COMP-EINT we use the zlib library.

Our prototype architecture is depicted in Figure 6. We modified the user space of EncFS to include the CMC cipher for block encryption and the new integrity algorithms. The server uses the underlying file system (i.e., reiserfs) for the storage of the encrypted files. The Merkle trees for integrity  $RTree_F$  and the index arrays of the random-looking blocks  $RArr_F$  are stored with the encrypted files in the untrusted storage space on the server. For faster integrity checking (in particular to improve the running time of the SearchOffset algorithm used in the Update and Check algorithms of the RAND-EINT and COMP-EINT constructions), we also keep the array  $RArr_F$  for each file, ordered by indices. The roots of the trees  $RTree_F$ , and the arrays  $IntStart_F$  and  $CtrVal_F$  or their hashes (if they are too large) are stored in a trusted storage space. In our current implementation, we use two extended attributes for each file  $F$ , one for the root of  $RTree_F$  and the second for the arrays  $IntStart_F$  and  $CtrVal_F$ , or their hashes.

By default, EncFS caches the last block content written to or read from the disk. In our implementation, we cached the last arrays  $RArr_F$ ,  $IntStart_F$  and  $CtrVal_F$  used in a block update or check operation. Since these arrays are typically small (a few hundred bytes), they easily fit into memory. We also evaluate the effect of caching of Merkle trees in our system in Section 7.1.

## 7 Performance Evaluation

In this section, we evaluate the performance of the new randomness test and compression integrity constructions for encrypted storage compared to that of Merkle trees. We ran our experiments on a 2.8 GHz Intel D processor machine with 1GB of RAM, running SuSE Linux 9.3 with kernel version 2.6.11. The hard disk used was an

80GB SATA 7200 RPM Maxtor.

The main challenge we faced in evaluating the proposed constructions was to come up with representative file system workloads. While the performance of the Merkle tree construction is predictable independently of the workload, the performance of the new integrity algorithms is highly dependent on the file contents accessed, in particular on the randomness of block contents. To our knowledge, there are no public traces that contain file access patterns, as well as the contents of the file blocks read and written. Due to the privacy implications of releasing actual users' data, we expect it to be nearly impossible to get such traces from a widely used system. However, we have access to three public NFS Harvard traces [9] that contain NFS traffic from several of Harvard's campus servers. The traces were collected at the level of individual NFS operations and for each read and write operation they contain information about the file identifier, the accessed offset in the file and the size of the request (but not the actual file contents).

To evaluate the integrity algorithms proposed in this paper, we perform two sets of experiments. In the first one, we strive to demonstrate how the performance of the new constructions varies for different file contents. For that, we use representative files from a Linux distribution installed on one of our desktop machines, together with other files from the user's home directory, divided into several file types. We identify five file types of interest: text, object, executables, images, and compressed files, and build a set of files for each class of interest. All files of a particular type are first encrypted and the integrity information for them is built; then they are decrypted and checked for integrity. We report the performance results for the files with the majority of blocks not random-looking (i.e., text, executable and object) and for those with mostly random-looking blocks (i.e., image and compressed). In this experiment, all files are written and read sequentially, and as such the access pattern is not a realistic one.

In the second set of experiments, we evaluate the effect of more realistic access patterns on the performance of the integrity schemes, using the NFS Harvard traces. As the Harvard traces do not contain information about actual file block contents written to the disks, we generate synthetic block contents for each block write request. We define two types of block contents: low-entropy and high-entropy, and perform experiments assuming that either all blocks are low-entropy or all are high-entropy. These extreme workloads represent the "best" and "worst"-case for the new algorithms, respectively. We also consider a "middle"-case, in which a block is random-looking with a 50% probability, and plot

the performance results of the new schemes relative to the Merkle tree integrity algorithm for the best, middle and worst cases.

## 7.1 The Impact of File Block Contents on Integrity Performance

**File sets.** We consider a snapshot of the file system from one of our desktop machines. We gathered files that belong to five classes of interest: (1) *text files* are files with extensions .txt, .tex, .c, .h, .cpp, .java, .ps, .pdf; (2) *object files* are system library files from the directory /usr/local/lib; (3) *executable files* are system executable files from directory /usr/local/bin; (4) *image files* are JPEG files and (5) *compressed files* are gzipped tar archives. Several characteristics of each set, including the total size, the number of files in each set, the minimum, average and maximum file sizes and the fraction of file blocks that are considered random-looking by the entropy test are given in Table 2.

**Experiments.** We consider three cryptographic file systems: (1) MT-EINT with CBC-AES for encrypting file blocks; (2) RAND-EINT with CMC encryption; (3) COMP-EINT with CBC-AES encryption. For each cryptographic file system, we first write the files from each set; this has the effect of automatically encrypting the files, and running the Update algorithm of the integrity method for each file block. Second, we read all files from each set; this has the effect of automatically decrypting the files, and running the Check algorithm of the integrity method for each file block. We use file blocks of size 4KB in the experiments.

**Micro-benchmarks.** We first present a micro-benchmark evaluation for the text and compressed file sets in Figure 7. We plot the total time to write and read the set of text and compressed files, respectively. The write time for a set of files includes the time to encrypt all the files in the set, create new files, write the encrypted contents in the new files and build the integrity information for each file block with algorithms Update and Append. The read time for a set of files includes the time to retrieve the encrypted files from disk, decrypt each file from the set and check the integrity of each file block with algorithm Check. We separate the total time incurred by the write and read experiments into the following components: encryption/decryption time (either AES or CMC); hashing time that includes the computation of both SHA-1 and HMAC; randomness check time (either the entropy test for RAND-EINT or compression/decompression time for COMP-EINT); Merkle tree operations (e.g., given a leaf index, find its index in inorder traversal or given an inorder index of

a node in the tree, find the inorder index of its sibling and parent); the time to update and check the root of the tree (the root of the Merkle tree is stored as an extended attribute for the file) and disk waiting time.

The results show that the cost of CMC encryption and decryption is about 2.5 times higher than that of AES encryption and decryption in CBC mode. Decompression is between 4 and 6 times faster than compression and this accounts for the good read performance of COMP-EINT.

A substantial amount of the MT-EINT overhead is due to disk waiting time (for instance, 39% at read for text files) and the time to update and check the root of the Merkle tree (for instance, 30% at write for compressed files). In contrast, due to smaller sizes of the Merkle trees in the RAND-EINT and COMP-EINT file systems, the disk waiting time and the time to update and check the root of the tree for text files are smaller. The results suggest that caching of the hash values stored in Merkle trees in the file system might reduce the disk waiting time and the time to update the root of the tree and improve the performance of all three integrity constructions, and specifically that of the MT-EINT algorithm. We present our results on caching next.

**Caching Merkle trees.** We implemented a global cache that stores the latest hashes read from Merkle trees used to either update or check the integrity of file blocks. As an optimization, when we verify the integrity of a file block, we compute all the hashes on the path from the node up to the root of the tree until we reach a node that is already in the cache and whose integrity has been validated. We store in the cache only nodes that have been verified and that are authentic. When a node in the cache is written, all its ancestors on the path from the node to the root, including the node itself, are evicted from the cache.

We plot the total file write and read time in seconds for the three cryptographic file systems as a function of different cache sizes. We also plot the average integrity bandwidth per block in a log-log scale. Finally, we plot the cumulative size of the untrusted storage  $US_F$  for all files from each set. We show the combined graphs for low-entropy files (text, object and executable files) in Figure 8 and for high-entropy files (compressed and image files) in Figure 9.

The results show that MT-EINT benefits mostly on reads by implementing a cache of size 1KB, while the write time is not affected greatly by using a cache. The improvements for MT-EINT using a cache of 1KB are as much as 25.22% for low-entropy files and 20.34% for high-entropy files in the read experiment. In the following, we compare the performance of the three constructions for the case in which a 1KB cache is used.

	Total size	No. files	Min. file size	Max. file size	Avg. file size	Fraction of random-looking blocks
Text	245 MB	808	27 bytes	34.94 MB	307.11 KB	0.0351
Objects	217 MB	28	15 bytes	92.66 MB	7.71 MB	0.0001
Executables	341 MB	3029	24 bytes	13.21 MB	112.84 KB	0.0009
Image	189 MB	641	17 bytes	2.24 MB	198.4 KB	0.502
Compressed	249 MB	2	80.44 MB	167.65 MB	124.05 MB	0.7812

Table 2: File set characteristics.

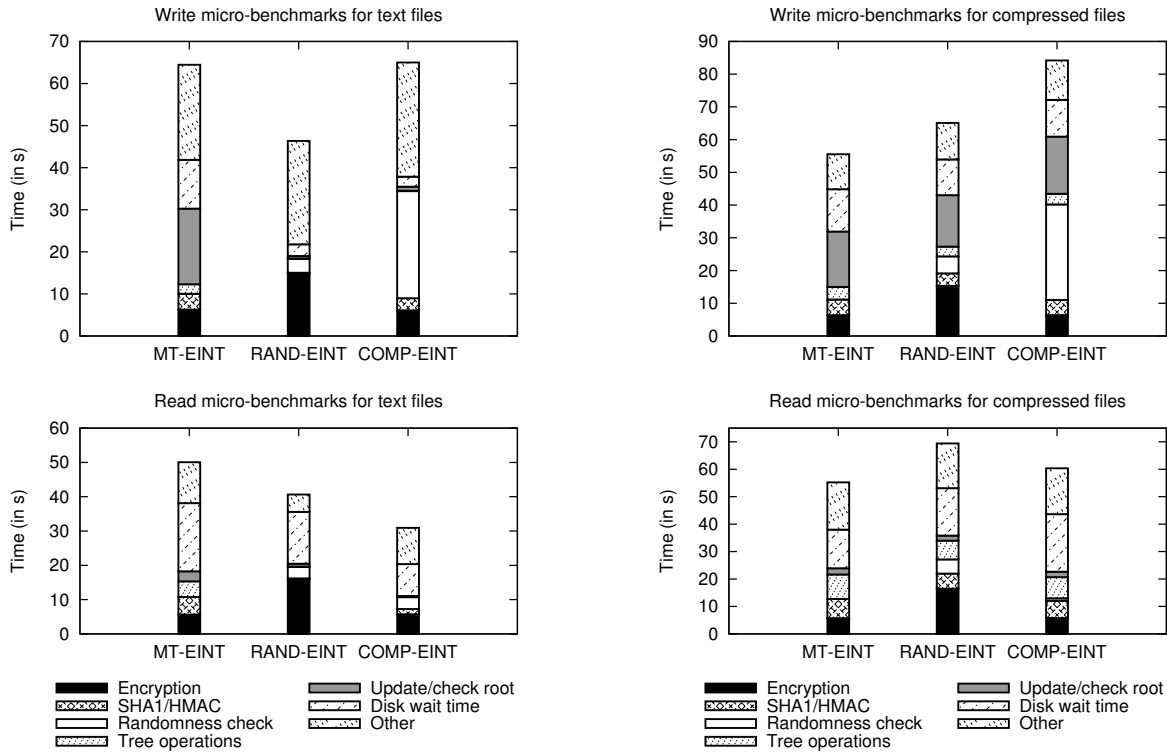


Figure 7: Micro-benchmarks for text and compressed files.

**Results for low-entropy files.** For sets of files with a low percent of random-looking blocks (text, object and executable files), RAND-EINT outperforms MT-EINT with respect to all the metrics considered. The performance of RAND-EINT compared to that of MT-EINT is improved by 31.77% for writes and 20.63% for reads. The performance of the COMP-EINT file system is very different in the write and read experiments due to the cost difference of compression and decompression. The write time of COMP-EINT is within 4% of the write time of MT-EINT and in the read experiment COMP-EINT outperforms MT-EINT by 25.27%. The integrity bandwidth of RAND-EINT and COMP-EINT is 92.93 and 58.25 times, respectively, lower than that of MT-EINT. The untrusted storage for integrity for RAND-EINT and COMP-EINT is reduced 2.3 and 1.17 times, respectively, compared to MT-EINT.

**Results for high-entropy files.** For sets of files with a high percent of random-looking blocks (image and compressed files), RAND-EINT adds a maximum performance overhead of 4.43% for writes and 18.15% for reads compared to MT-EINT for a 1KB cache. COMP-EINT adds a write performance overhead of 38.39% compared to MT-EINT, and performs within 1% of MT-EINT in the read experiment. The average integrity bandwidth needed by RAND-EINT and COMP-EINT is lower by 30.15% and 10.22%, respectively, than that used by MT-EINT. The untrusted storage for integrity used by RAND-EINT is improved by 9.52% compared to MT-EINT and that of COMP-EINT is within 1% of the storage used by MT-EINT. The reason that the average integrity bandwidth and untrusted storage for integrity are still reduced in RAND-EINT compared to MT-EINT is that in the set of high-entropy

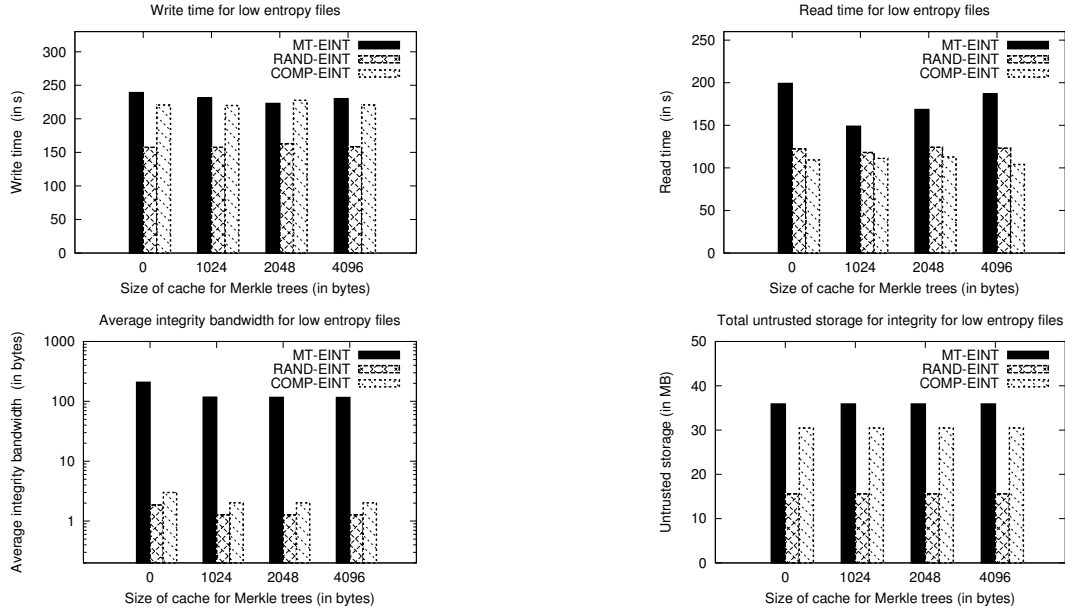


Figure 8: Evaluation for low-entropy files (text, object and executable files).

files considered only about 70% of the blocks have high entropy. We would expect that for files with 100% high-entropy blocks, these two metrics will exhibit a small overhead with both RAND-EINT and COMP-EINT compared to MT-EINT (this is actually confirmed in the experiments from the next section). However, such workloads with 100% high entropy files are very unlikely to occur in practice.

## 7.2 The Impact of File Access Patterns on Integrity Performance

**File traces.** We considered a subset of the three NFS Harvard traces [9] (LAIR, DEASNA and HOME02), each collected during one day. We show several characteristics of each trace, including the number of files and the total number of block write and read operations, in Table 3. The block size in these traces is 4096 bytes and we have implemented a 1KB cache for Merkle trees.

	Number of files	Number of writes	Number of reads
LAIR	7017	66331	23281
DEASNA	890	64091	521
HOME02	183	89425	11815

Table 3: NFS Harvard trace characteristics.

**Experiments.** We replayed each of the three traces with three types of block contents: all low-entropy, all high-entropy and 50% high-entropy. For each experiment, we measured the total running time, the average

integrity bandwidth and the total untrusted storage for integrity for RAND-EINT and COMP-EINT relative to MT-EINT and plot the results in Figure 10. We represent the performance of MT-EINT as the horizontal axis in these graphs and the performance of RAND-EINT and COMP-EINT relative to MT-EINT. The points above the horizontal axis are overheads compared to MT-EINT, and the points below the horizontal axis represent improvements relative to MT-EINT. The labels on the graphs denote the percent of random-looking blocks synthetically generated.

**Results.** The performance improvements of RAND-EINT and COMP-EINT compared to MT-EINT are as high as 56.21% and 56.85%, respectively, for the HOME02 trace for low-entropy blocks. On the other hand, the performance overhead for high-entropy blocks are at most 54.14% for RAND-EINT (in the LAIR trace) and 61.48% for COMP-EINT (in the DEASNA trace). RAND-EINT performs better than COMP-EINT when the ratio of read to write operations is small, as is the case for the DEASNA and HOME02 trace. As this ratio increases, COMP-EINT outperforms RAND-EINT.

For low-entropy files, both the average integrity bandwidth and the untrusted storage for integrity for both RAND-EINT and COMP-EINT are greatly reduced compared to MT-EINT. For instance, in the DEASNA trace, MT-EINT needs 215 bytes on average to update or check the integrity of a block, whereas RAND-EINT and COMP-EINT only require on average 0.4 bytes.

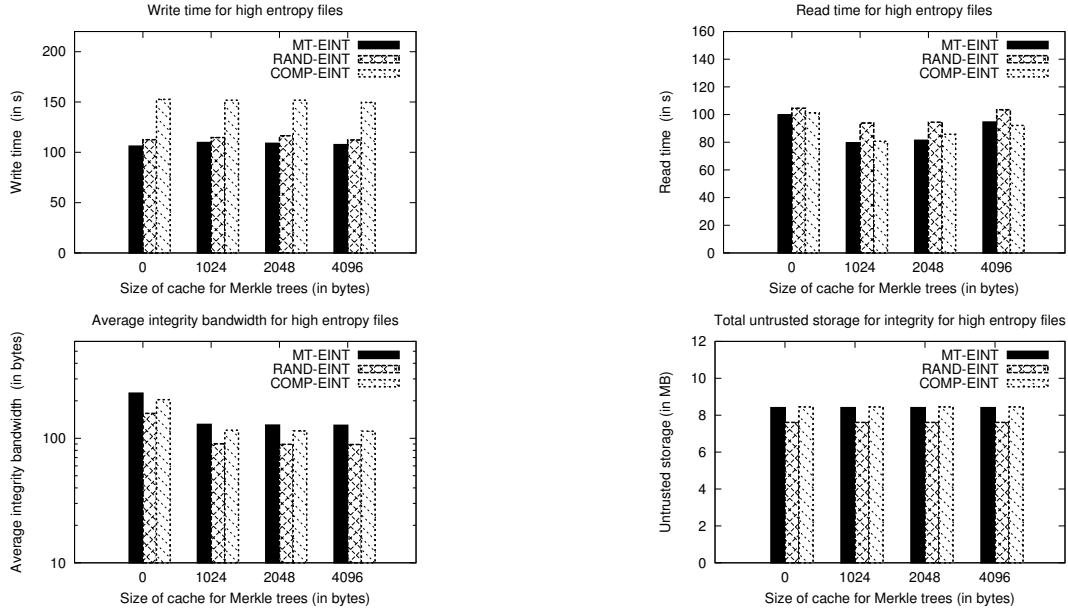


Figure 9: Evaluation for high-entropy files (image and compressed files).

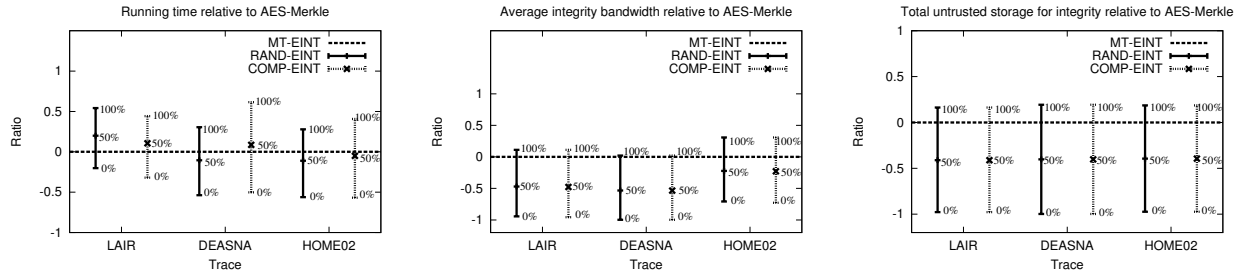


Figure 10: Running time, average integrity bandwidth and storage for integrity of RAND-EINT and COMP-EINT relative to MT-EINT. Labels on the graphs represent percentage of random-looking blocks.

The amount of additional untrusted storage for integrity in the DEASNA trace is 2.56 MB for MT-EINT and only 7 KB for RAND-EINT and COMP-EINT. The maximum overhead added by both RAND-EINT and COMP-EINT compared to MT-EINT for high-entropy blocks is 30.76% for the average integrity bandwidth (in the HOME02 trace) and 19.14% for the amount of untrusted storage for integrity (in the DEASNA trace).

### 7.3 Discussion

From the evaluation of the three constructions, it follows that none of the schemes is a clear winner over the others with respect to all the four metrics considered. Since the performance of both RAND-EINT and COMP-EINT is greatly affected by file block contents, it would be beneficial to know the percentage of high-entropy blocks in

practical filesystem workloads. To determine statistics on file contents, we have performed a user study on several machines from our department running Linux. For each user machine, we have measured the percent of high-entropy blocks and the percent of blocks that cannot be compressed enough from users' home directories. The results show that on average, 28% percent of file blocks have high entropy and 32% percent of file blocks cannot be compressed enough to fit a MAC inside.

The implications of our study are that, for cryptographic file systems that store files similar to those in users' home directories, the new integrity algorithms improve upon Merkle trees with respect to all four metrics of interest. In particular, COMP-EINT is the best option for primarily read-only workloads when minimizing read latency is a priority, and RAND-EINT is the best

choice for most other workloads. On the other hand, for an application in which the large majority of files have high-entropy (e.g., a file sharing application in which users transfer mostly audio and video files), the standard MT-EINT still remains the best option for integrity. We recommend that all three constructions be implemented in a cryptographic file system. An application can choose the best scheme based on its typical workload.

The new algorithms that we propose can be applied in other settings in which authentication of data stored on untrusted storage is desired. One example is checking the integrity of arbitrarily-large memory in a secure processor using only a constant amount of trusted storage [5, 7]. In this setting, a trusted checker maintains a constant amount of trusted storage and, possibly, a cache of data blocks most recently read from the main memory. The goal is for the checker to verify the integrity of the untrusted memory using a small bandwidth overhead.

The algorithms described in this paper can be used only in applications where the data that needs to be authenticated is encrypted. However, the COMP-EINT integrity algorithm can be easily modified to fit into a setting in which data is only authenticated and not encrypted, and can thus replace Merkle trees in such applications. On the other hand, the RAND-EINT integrity algorithm is only suitable in a setting in which data is encrypted with a tweakable cipher, as the integrity guarantees of this algorithm are based on the security properties of such ciphers.

## 8 Related Work

We have focused on Merkle trees as our point of comparison, though there are other integrity protections used on various cryptographic file systems that we have elided due to their greater expense in various measures. For example, a common integrity method used in cryptographic file systems such as TCFS [6] and SNAD [25] is to store a hash or message authentication code for each file block for authenticity. However, these approaches employ trusted storage linear in the number of blocks in the file (either the hashes or a counter per block). In systems such as SFS [22], SFSRO [11], Cepheus [10], FARSITE [1], Plutus [17], SUNDR [20] and IBM StorageTank [23, 27], a Merkle tree per file is built and the root of the tree is authenticated (by either digitally signing it or storing it in a trusted meta-data server). In SiR-iUS [13], each file is digitally signed for authenticity, and so in addition the integrity bandwidth to update or check a block in a file is linear in the file size. Tripwire [19] is a user-level tool that computes a hash per file and stores it in trusted storage. While this approach achieves constant

trusted storage for integrity per file, the integrity bandwidth is linear in the number of blocks in the file. For journaling file systems, an elegant solution for integrity called *hash logging* is provided by PFS [32]. The hashes of file blocks together with the file system metadata are stored in the file system log, a protected memory area. However, in this solution the amount of trusted storage for integrity for a file is linear in the number of blocks in the file.

Riedel et al. [28] provides a framework for extensively evaluating the security of storage systems. Wright et al. [33] evaluates the performance of five cryptographic file systems, focusing on the overhead of encryption. Two other recent surveys about securing storage systems are by Sivathanu et al. [31] and Kher and Kimand [18].

## 9 Conclusion

We have proposed two new integrity constructions, RAND-EINT and COMP-EINT, that authenticate file blocks in a cryptographic file system using only a constant amount of trusted storage per file. Our constructions exploit the typical low entropy of block contents and sequentiality of file block writes to reduce the additional costs of integrity protection. We have evaluated the performance of the new constructions relative to the widely used Merkle tree algorithm, using files from a standard Linux distribution and NFS traces collected at Harvard university. Our experimental evaluation demonstrates that the performance of the new algorithms is greatly affected by file block contents and file access patterns. For workloads with majority low-entropy file blocks, the new algorithms improve upon Merkle trees with respect to all the four metrics considered.

## References

- [1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. 5th Symposium on Operating System Design and Implementation (OSDI)*. Usenix, 2002.
- [2] Advanced encryption standard. Federal Information Processing Standards Publication 197, U.S. Department of Commerce/National Institute of Standards and Technology, National Technical Information Service, Springfield, Virginia, Nov. 2001.
- [3] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Proc. Crypto 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1996.

- [4] J. Black and H. Urtubia. Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption. In *Proc. 11th USENIX Security Symposium*, pages 327–338, 2002.
- [5] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 1994.
- [6] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for Unix. In *Proc. USENIX Annual Technical Conference 2001, Freenix Track*, pages 199–212, 2001.
- [7] D. E. Clarke, G. E. Suh, B. Gassend, A. Sudan, M. van Dijk, and S. Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *Proc. 26th IEEE Symposium on Security and Privacy*, pages 139–153, 2005.
- [8] DES modes of operation. Federal Information Processing Standards Publication 81, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, 1980.
- [9] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proc. Second USENIX Conference on File and Storage Technologies (FAST)*, pages 203–216, 2003.
- [10] K. Fu. Group sharing and random access in cryptographic storage file systems. Master’s thesis, Massachusetts Institute of Technology, 1999.
- [11] K. Fu, F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20:1–24, 2002.
- [12] FUSE: filesystem in userspace. <http://fuse.sourceforge.net>.
- [13] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiR-iUS: Securing remote untrusted storage. In *Proc. Network and Distributed Systems Security (NDSS) Symposium 2003*, pages 131–145. ISOC, 2003.
- [14] V. Gough. EncFS encrypted filesystem. <http://arg0.net/wiki/encfs>, 2003.
- [15] S. Halevi and P. Rogaway. A tweakable enciphering mode. In *Proc. Crypto 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 482–499. Springer-Verlag, 2003.
- [16] S. Halevi and P. Rogaway. A parallelizable enciphering mode. In *Proc. The RSA conference - Cryptographer’s track (RSA-CT)*, volume 2964 of *Lecture Notes in Computer Science*, pages 292–304. Springer-Verlag, 2004.
- [17] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. Second USENIX Conference on File and Storage Technologies (FAST)*, 2003.
- [18] V. Kher and Y. Kim. Securing distributed storage: Challenges, techniques, and systems. In *Proc. First ACM International Workshop on Storage Security and Survivability (StorageSS 2005)*, 2005.
- [19] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A filesystem integrity checker. In *Proc. Second ACM Conference on Computer and Communication Security (CCS)*, pages 18–29, 1994.
- [20] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository. In *Proc. 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 121–136. Usenix, 2004.
- [21] M. Liskov, R. Rivest, and D. Wagner. Tweakable block ciphers. In *Proc. Crypto 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, 2002.
- [22] D. Mazieres, M. Kaminsky, M. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 124–139. ACM Press, 1999.
- [23] J. Menon, D. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM Storage Tank - a heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2), 2003.
- [24] R. Merkle. A certified digital signature. In *Proc. Crypto 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer-Verlag, 1989.
- [25] E. Miller, D. Long, W. Freeman, and B. Reed. Strong security for distributed file systems. In *Proc. the First USENIX Conference on File and Storage Technologies (FAST)*, 2002.
- [26] A. Oprea, M. K. Reiter, and K. Yang. Space-efficient block storage integrity. In *Proc. Network and Distributed System Security Symposium (NDSS)*. ISOC, 2005.
- [27] R. Pletka and C. Cachin. Cryptographic security for a high-performance distributed file system. Technical Report RZ 3661, IBM Research, Sept. 2006.
- [28] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. In *Proc. First USENIX Conference on File and Storage Technologies (FAST)*, pages 15–30, 2002.
- [29] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Proc. 11th International Workshop on Fast Software Encryption (FSE 2004)*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer-Verlag, 2004.
- [30] Secure hash standard. Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/National Institute of Standards and Technology, National Technical Information Service, Springfield, Virginia, Apr. 1995.
- [31] G. Sivathanu, C. Wright, and E. Zadok. Ensuring data integrity in storage: Techniques and applications. In *Proc. ACM Workshop on Storage Security and Survivability*, 2005.
- [32] C. A. Stein, J. Howard, and M. I. Seltzer. Unifying file system protection. In *Proc. USENIX Annual Technical Conference*, pages 79–90, 2001.
- [33] C. P. Wright, J. Dave, and E. Zadok. Cryptographic file systems performance: What you don’t know can hurt you. In *Proc. Second Intl. IEEE Security in Storage Workshop (SISW)*, 2003.

# ***Discoverer: Automatic Protocol Reverse Engineering from Network Traces***

Weidong Cui  
Microsoft Research  
wdcui@microsoft.com

Jayanthkumar Kannan  
UC Berkeley  
kjk@cs.berkeley.edu

Helen J. Wang  
Microsoft Research  
helenw@microsoft.com

## **Abstract**

Application-level protocol specifications are useful for many security applications, including intrusion prevention and detection that performs deep packet inspection and traffic normalization, and penetration testing that generates network inputs to an application to uncover potential vulnerabilities. However, current practice in deriving protocol specifications is mostly manual. In this paper, we present *Discoverer*, a tool for automatically reverse engineering the protocol message formats of an application from its network trace. A key property of *Discoverer* is that it operates in a protocol-independent fashion by inferring protocol idioms commonly seen in message formats of many application-level protocols. We evaluated the efficacy of *Discoverer* over one text protocol (HTTP) and two binary protocols (RPC and CIFS/SMB) by comparing our inferred formats with true formats obtained from Ethereal [5]. For all three protocols, more than 90% of our inferred formats correspond to exactly one true format; one true format is reflected in five inferred formats on average; our inferred formats cover over 95% of messages, which belong to 30-40% of true formats observed in the trace.

## **1 Introduction**

Application-level protocol specifications are useful for many security applications. Penetration testing can leverage protocol specifications to generate network inputs to an application to uncover potential vulnerabilities. For network management, protocol specifications can also be used to identify protocols and tunnelings in monitored network traffic. Generic protocol analyzers (GAPA [1] and binpac [16]) are important mechanisms for intrusion detection or firewall systems to perform deep packet inspection. These analyzers take protocol specifications as

input for their analyses.

To date, protocol specifications for the above applications are specified from documentation or reverse engineered manually. Such efforts are painstakingly time-consuming and error-prone. It took the open-source SAMBA project 12 years to manually reverse engineer the Microsoft SMB protocol [18]. In another example, the Yahoo messenger protocol has also been persistently reverse engineered, despite which, the open source clients [6] regularly require patching to support proprietary changes in the Yahoo protocol. Sometimes, the period between the availability of an official client and an open-source client has been a month, with some open-source projects simply abandoning the effort due to the frequent changes initiated by Yahoo.

To address this pain, we tackle the problem of automatic protocol reverse engineering. There can be two sources of given input for the reverse-engineering task: network traces and application code. In this paper, we present our tool, *Discoverer*, which performs automatic reverse engineering from network traces. We leave application-code-based reverse engineering as future work.

In *Discoverer*, we focus on reverse engineering the message format specification and leave the protocol state machine inference to our future work. To automatically reverse engineer message formats for a wide range of protocols, we face three main challenges: (1) We have very few hints from the network trace. The only evident information from the trace is the directionality of byte streams. (2) Protocols are significantly different from each other. (3) Protocol message formats are often context-sensitive where earlier fields dictate the parsing of the subsequent part of the message.

To make our tool general, we base our design on inferring protocol idioms commonly seen in message formats of many protocols. To cope with the few hints, we dissect

the formless byte streams into text and binary segments or tokens as a starting point for clustering messages with similar patterns, where each cluster approximates a message format. By comparing messages in a cluster and observing the characteristics of known cross-field dependencies (such as a length field followed by a string of the length), we infer additional properties for the tokens, which in turn can be leveraged to refine and divide the clusters of messages, where each subcluster approximates a more precise format. This process continues recursively until we can no longer divide up any message clusters based on the newly finished inference. After this recursive clustering phase, we look at all message clusters globally through a type-based sequence alignment algorithm, and merge similar clusters into one. This way, we can produce more concise message formats.

We have evaluated Discoverer over traces of a representative set of protocols consisting of one text protocol (HTTP) and two binary protocols (RPC and CIFS/SMB). We calibrated our design over some of these traces, and used the remaining for validation. The three main metrics for our tool are *correctness* (“does one inferred format correspond to exactly one true format?”), *conciseness* (“how many inferred formats is a single true format reflected in?”), and *coverage* (“how many messages are covered by the inferred formats?”). Across all protocols we tested, more than 90% inferred formats correspond to exactly one true format; one true format is reflected in five inferred formats on average; our inferred formats cover over 95% messages, which belong to 30-40% of true formats observed in the trace. Such significant difference between message and format coverage is due to the heavy-tail distribution of message format popularity commonly seen in practice.

Although our reverse-engineered message formats are imperfect, we anticipate them to be still practical for the aforementioned applications. For instance, penetration testing guided by our reverse-engineered formats is likely to be much more effective than that with random inputs. Protocol fingerprinting and tunneling detection probably do not require perfect protocol specifications. For applications like firewalls which would err with imperfect specifications, our tool could still serve as a help to ease the manual protocol specification process.

We organize the rest of the paper as follows. We discuss common protocol idioms and the scope of Discoverer in Section 2. We describe the design of Discoverer in detail in Section 3. We present our evaluation methodology and results in Section 4. We discuss related work in Section 5, and limitations and future work in Section 6. Finally, we summarize the paper in Section 7.

## 2 Problem Statement

Many application-level protocols share common protocol idioms which correspond to the essential components in a protocol specification. To make our reverse-engineering algorithm applicable to many protocols, we base our design on inferring the common protocol idioms. In this section, we first describe these idioms and then explain the scope of Discoverer.

### 2.1 Common Protocol Idioms

Most application-level protocols involve the concept of an *application session*, which consists of a series of *messages* (also known as Application-level Data Units or ADUs) between two hosts that accomplishes a specific task. The structure of an application session is determined by the application’s *protocol state machine*, an essential component in a protocol specification that characterizes all possible legitimate sequences of messages. The structure of an application message is determined by the application’s *message format specification*, another essential component in a protocol specification. A message format specifies a sequence of *fields* and their semantics. Common field semantics include *length* (reflecting the size of a subsequent field with a variable length), *offset* (determining the byte offset of another field from a certain point like the start of the message), *pointer* (a special offset that specifies the index of a field in an array of arbitrary items), *cookie* (session-specific opaque data that appears in messages from both sides of the application session; session IDs are an example of cookie fields), *endpoint-address* (encoding IP addresses or port numbers in some form), and *set* (a group of fields that can be put in an arbitrary order).

One particular type of fields is the *Format Distinguisher* (FD) field. The value of this field serves to differentiate the format of the subsequent part of the message, which reflects the context-sensitive nature in the grammar of many application-level protocols. A message may have a sequence of FD fields, particularly when multiple protocols are encapsulated. For instance, a CIFS/SMB message consists of a NetBIOS header encapsulating an SMB header, which in turn may encapsulate a RPC message. This implies that the applications need to scan a message from *left-to-right*, decoding a FD field before parsing the subsequent part of the message.

### 2.2 Scope of Discoverer

In this paper, we focus on deriving the message format specification and leave protocol state machine inference

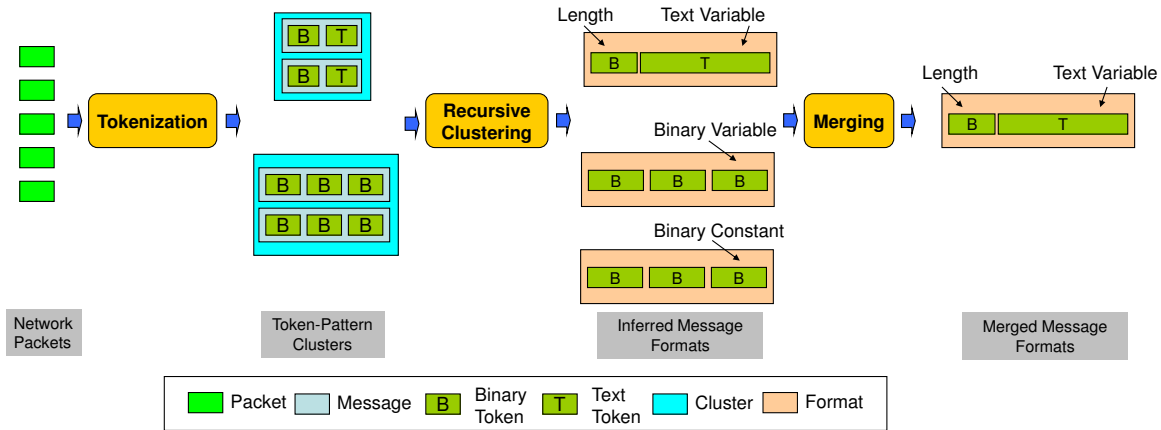


Figure 1: Overview of Discoverer’s architecture. In the example, we assume there is a single true message format which has two fields: the first binary field of a single byte represents the length of the second text field. There are two token patterns because, when the text field is shorter than a threshold, it is treated as binary. In the merging phase, this kind of tokenization errors is corrected.

to our future work. We assume synchronous protocols to identify message boundaries. A message is a consecutive chunk of application-level data sent in one direction. It spans one or more packets in a TCP or UDP connection, where a UDP connection is a pair of unidirectional UDP flows that are matched on source/destination IP address/port number. We only aim to deal with applications that do not obfuscate their payloads. We do not aim to capture timing semantics (e.g., “message 1 usually follows message 2 within 10 seconds”).

### 3 Design

In this section, we first present an overview of Discoverer, then describe the three main phases of Discoverer in detail, and finally give a concrete example of a message format inferred by Discoverer.

#### 3.1 Overview

The basic idea of Discoverer is to cluster messages with the same format together and infer the message format by comparing messages in a single cluster. We achieve this in three main phases (illustrated in Figure 1).

- **Tokenization and Initial Clustering:** This phase operates on the raw packets, and helps in identifying field boundaries in a message and giving the first order structure to the unlabeled messages. We first re-assemble the packets into messages, and then break up a message into a sequence of tokens which is an

approximation to a sequence of fields. Tokens belong to one of two *token classes*: binary or text. We then classify messages into various clusters based on each message’s token pattern, which is simply represented by the message direction and classes of its tokens.

- **Recursive Clustering:** Since messages with the same token pattern do not necessarily have the same format, this phase further divides clusters of messages so that messages in each cluster have the same format, and infers the message format by comparing messages in each single cluster. To do so, we mimic the left-to-right recursive parsing of applications processing messages by recursively repeating the following steps. We first infer the message format that captures the content of all messages in a cluster. Then we identify the first FD field (which decides the format of the subsequent part of the message) in a left-to-right scan and use the values of this FD field to divide the cluster into subclusters.
- **Merging:** This phase mitigates the over-classification problem, namely, messages of the same format may be scattered into multiple clusters. To do so, we merge similar message formats by using a *type*-based sequence alignment algorithm that compares the field structure of two inferred message formats.

A key design rationale for Discoverer is to be *conservative*: it may scatter messages of the same format into more than one cluster, but it should not collate messages of different formats into the same cluster. This rationale is to ensure the correctness of inferred formats because, if there are messages of more than one format in a cluster, the inferred format might be too general by trying to capture multiple message formats at once.

## 3.2 Tokenization and Initial Clustering

### 3.2.1 Tokenization

A token is a sequence of consecutive bytes likely to belong to the same application-level field. We require that the tokenization process works *without* any particular distinction between text and binary protocols, since our tool is intended to be fully automatic and we wish to spare the user from the manual effort required to distinguish between text and binary protocols. Further, it is hard to declare a protocol as purely text or purely binary, since text protocols can contain binary bytes (e.g., an image file transferred over HTTP) and most binary protocols contain a few text fields (e.g., the name of a file).

Our tokenization procedure generates two *classes* of tokens: text and binary. A text token is intended to span the several bytes of a single message field representing some text (such as “GET” in an HTTP request). Our procedure for finding text tokens is as follows: we first identify text bytes by comparing them with the ASCII values of printable characters, and then consider a sequence of text bytes sandwiched between two binary bytes as a text segment. To avoid mistaking binary bytes for text bytes, we require this sequence to have a minimum length. Then we use a set of delimiters (e.g., space and tab) to divide a text segment into tokens. We also look for Unicode encodings in messages. For binary fields, identifying field boundaries is very hard; so we instead simply declare a single binary byte to be a binary token in its own right. Note that this procedure can admit errors: consecutive binary bytes with ASCII values of printable characters are wrongly marked as a text token; a text string shorter than the minimum length is wrongly marked as binary tokens; a text field consisting of some white space characters is wrongly divided into multiple text tokens. We correct this kind of errors in the merging phase (see Section 3.4).

### 3.2.2 Initial Clustering by Token Patterns

Byte-wise sequence alignment based on the Needleman-Wunsch algorithm [15] has been used

in previous studies [3, 11, 17] for aligning and comparing messages. We find that byte-wise sequence alignment, while ideally suited to align messages with similar *byte patterns*, is not suitable for aligning messages with the same *format*. For instance, fields with variable lengths may lead to mis-alignment of two messages of the same format. Further, parameter selection for sequence alignment is also hard as shown in [3].

To avoid aligning messages, we cluster messages based on their token patterns. The token pattern assigned to a message is a tuple,  $(dir, class\_of\_token\_1, class\_of\_token\_2, \dots)$ , where *dir* is the direction of the message (client to server or vice versa), followed by the classes of all tokens in the message. We consider the message direction because messages in opposite directions tend to have different formats. An example of a token pattern is  $(client\_to\_server, text, binary, text)$ .

Note that this initial clustering is coarse-grained since messages with different formats may have the same token pattern. For instance, SMTP commands typically have two text tokens (“MAIL receiver”, “RCPT sender”, “HELO server-name”). In the recursive clustering phase, we improve the granularity of this clustering by recursively identifying FD tokens and dividing clusters.

## 3.3 Recursive Clustering

Our recursive clustering relies on identifying format distinguisher (FD) tokens. To find FD tokens, we need to invoke both format inference and format comparison. In this section, we first explain these procedures before describing how we recursively identify FD tokens and divide clusters.

### 3.3.1 Format Inference

The format inference phase takes as input a set of messages and infers a format that succinctly captures the content of the set of messages. Our inferred *message format* is defined to be a sequence of token specifications which include not only *token semantics* but also *token properties*. We introduce token properties because we cannot infer the semantic meaning for every token and certain token properties are useful for describing the message format. Token properties currently cover two perspectives: *binary* vs. *text* and *constant* vs. *variable*. The first property reflects the token class, and the second decides if a token takes the same value across all messages of the same format (i.e., constant token) or different values in different application sessions (i.e., variable

token). We also define the *type* of a token to be the sum of its semantic and property.

We now describe how token properties and semantics are derived.

*Property Inference:* Token class is already identified during the tokenization phase. Constant or variable tokens can also be easily identified. Since the set of messages come from a single token-pattern cluster, tokens in one message can be directly compared against their counterparts in another message by simply using the token offset. Thus, constant tokens are those that take the same value across the entire set of messages, and variable tokens are those that take more than one value.

*Semantic Inference:* We currently support three semantics: length, offset, and cookie (see Section 2.1 for definitions). We will discuss how it may be possible to support other semantics in Section 6. We identify cookie fields at the end of the merging phase since it requires correlating multiple messages in the same session. We employ the heuristics in RolePlayer [3] for doing this. Our heuristics for identifying length and offset fields are an extension of those in RolePlayer. The intuition for identifying length fields is that, for a specific pair of messages, the *difference* in the values of potential length fields (at most four consecutive binary tokens or a text token in the decimal or hex format) reflects the *difference* of the sizes of the messages or some subsequent tokens. We thus simply check for a match between the value difference and the size difference. If a match holds for all pairs of messages in the cluster, the potential length field is declared to be a length field. For offset fields, we compare the value difference with the difference of the offsets of some subsequent tokens.

### 3.3.2 Format Comparison

The goal of this procedure is to decide if two inferred message formats are the same. Given two formats, it scans these two formats token-by-token from left-to-right and matches the inferred type (i.e., semantic and property) of a token from one format against its counterpart from the other. If all tokens match, the two formats are considered to be the same.

Ideally, two tokens can be considered to match if their semantics match. However, since there are always tokens that we do not have semantics for, we need to compare their values (they have the same token class since the two formats have the same token pattern). We allow a constant token to match with a variable token if the latter takes the value of the former at least once. We also allow a variable token to match with another if there is an overlap in the two sets of values taken by them. Note that

these policies are conservative, which is in line with our design rationale.

### 3.3.3 Recursive Clustering by Format Distinguishers

We identify FD tokens with the following algorithm. First, we invoke format inference on the set of messages in a cluster. Then, we scan the format token by token from left to right to identify FD tokens. We use three criteria in determining if a token is a FD:

1. We first check if the number of unique values taken by this token across the set of messages is less than a threshold, referred to as the maximum distinct values for a FD token. This is because a FD token typically takes a few values corresponding to the number of different formats.
2. For tokens satisfying the first criterion, we perform a second test as follows. The cluster is divided into subclusters, one for each unique value taken by this token. Each subcluster consists of messages where the candidate FD token takes a specific value. We then require that the size of the largest subcluster exceeds a threshold, referred to as the minimum cluster size. This is to guarantee that we can make a meaningful format inference in at least one subcluster. Otherwise, we gain nothing by continuing this splitting.
3. If the potential FD token passes the second criterion, we invoke format comparison across subclusters to see if their formats are different from each other. We then merge those that manifest the same formats and leave others intact.

This process is recursively performed on each of the subclusters because a message may have more than one FD token. We find the next FD token by scanning further down the message towards the right (end) of the message. It is necessary to scan all the way to the end because we need to recognize all FDs to obtain a good clustering and format inference.

When looking for the next FD token, the format inference is invoked again on the set of messages in each subcluster. This is because the inferred token properties and semantics might change because the set of messages has become smaller, and it is possible for stronger properties to hold. For instance, a previously variable token might now be a constant token; a previously variable token might now be identified as a length field.

### 3.4 Merging with Type-Based Sequence Alignment

In the tokenization and recursive clustering phases, we are conservative to ensure that the format inference procedure operates correctly on a set of messages of the same format. However, this leads to a new problem of over-classification, namely, messages of the same format may be scattered into more than one cluster. This problem can be quite severe; for instance, over a CIFS/SMB trace of almost four million messages, there are about 7,000 clusters/formats as input to this phase, while the total number of true formats is 130. The goal of the merging process is to coalesce similar formats from different clusters into a single one.

The key observation behind our merging phase is that, while sequence alignment [15] cannot be used for clustering messages of the same format, it can be used to align *formats* for identifying similar ones across different clusters. This is because we can leverage the rich token types (i.e., semantics and properties) inferred in the recursive clustering phase. For instance, knowing that a particular token is a length field in a format necessitates that its counterpart in another format is also a length field for these two formats to be considered a match. We refer to our algorithm for aligning formats as *type-based* sequence alignment.

In our type-based sequence alignment, we only allow two tokens of the same class (binary or text) to align with each other. We claim two aligned tokens are matched if they either have the same semantic or share at least one value (see Section 3.3.2 for details).

To compensate for tokenization errors, we allow gaps in our type-based sequence alignment. In addition to using gap penalties to control gaps, we introduce extra constraints to avoid excessive gaps. First, consecutive binary tokens in one message format are allowed to align with gaps if they precede or follow a text token in the other message format in the alignment, and the number of binary tokens is at most the size of the text token if the text token is aligned with a gap, or the size difference if it is aligned with another text token. This constraint is for handling the case of mistaking a sequence of binary tokens to be a text token or vice versa. Second, a text token is allowed to align with a gap, but we allow at most two gaps of this kind. This constraint is for handling the case that a text field consisting of some white space characters is mistakenly divided into multiple tokens.

When we align and compare two message formats to decide whether to merge them, we first check if the gap constraints can be satisfied. If no, we stop and claim the two formats are mismatched; otherwise, we continue to

check the number of mismatches. If there is at most one pair of aligned tokens mismatched, we claim the two formats are matched and merge them. Note that this is conservative because the mismatched token can be treated as a variable token that takes values from a new set covering both formats.

Since we use the gap constraints and the number of mismatches to decide whether to merge two message formats, our merging performance is insensitive to sequence alignment parameters—scores for match, mismatch and gap.

### 3.5 An Example

For better understanding, here we present a concrete example based on the SMB “Tree Connect AndX Request” message format to explain the design and output of Discoverer. We obtain the true message format from Ethereum (see Figure 2 and Figure 3). The final inferred format by Discoverer is shown in Table 1.

We can see that the inferred format is a sequence of tokens with token properties (binary vs. text, constant vs. variable) and semantics (e.g., length fields). For tokens with unknown semantics, their possible values are also taken into account in the format. Before the merging step, messages of this true format were scattered into 24 clusters in 18 different token patterns. Different token patterns are due to the “smb.signature” field. Since this field may take any random values, we will have a different token pattern when more than three consecutive bytes at a different offset take values from the printable ASCII range and are wrongly treated as a text token. Messages in some token patterns were further split into fine-grained clusters in the recursive clustering phase due to our conservative approach. Our merging technique mitigates this over-classification problem effectively. At the end, all of the 24 clusters were merged into a single one.

This example also shows the possibility of imprecise field boundaries. For example, the first null byte of the field “smb.nt.status” was treated as the null terminator for the text token before it. However, we believe this kind of imprecision will not affect the effectiveness of the inferred format but instead create some extra inferred formats with different values for “smb.nt.status”.

## 4 Evaluation

We implemented Discoverer in 5,700 lines of C++ code on Windows. The tool takes a network capture file either in the libpcap [12] or Netmon [14] format as input and outputs inferred message formats: a sequence of tokens with the inferred properties and semantics. Our

```

<proto name="nbss" showname="NetBIOS Session Service" size="4" pos="54">
  <field name="nbss.type" showname="Message Type: Session message" size="1" pos="54" show="0" value="00"/>
  <field show="Length: 156" size="3" pos="55" value="00009c"/>
</proto>
<proto name="smb" showname="SMB (Server Message Block Protocol)" size="156" pos="58">
  <field show="SMB Header" size="32" pos="58">
    <field show="Server Component: SMB" size="4" pos="58" value="ff534d42"/>
    <field name="smb.cmd" showname="SMB Command: Tree Connect AndX (0x75)" size="1" pos="62" show="0x75"
value="75"/>
    <field name="smb.nt_status" showname="NT Status: STATUS_SUCCESS (0x00000000)" size="4" pos="63"
show="0x00000000" value="00000000"/>
    <field show="Flags: 0x18" size="1" pos="67" value="18">
    <field show="Flags2: 0xc807" size="2" pos="68" value="07c8">
    <field name="smb.pid.high" showname="Process ID High: 0" size="2" pos="70" show="0" value="0000"/>
    <field name="smb.signature" showname="Signature: 05A09637B7419166" size="8" pos="72"
show="05a0:96:37:b7:41:91:66" value="05a09637b7419166"/>
    <field name="smb.reserved" showname="Reserved: 0000" size="2" pos="80" show="00:00" value="0000"/>
    <field name="smb.tid" showname="Tree ID: 0" size="2" pos="82" show="0" value="0000"/>
    <field name="smb.pid" showname="Process ID: 65279" size="2" pos="84" show="65279" value="fffe"/>
    <field name="smb.uid" showname="User ID: 2048" size="2" pos="86" show="2048" value="0008"/>
    <field name="smb.mid" showname="Multiplex ID: 128" size="2" pos="88" show="128" value="8000"/>
  </field>
  <field show="Tree Connect AndX Request (0x75)" size="124" pos="90">
    <field name="smb.wct" showname="Word Count (WCT): 4" size="1" pos="90" show="4" value="04"/>
    <field show="AndXCommand: No further commands (0xff)" size="1" pos="91" value="ff"/>
    <field name="smb.reserved" showname="Reserved: 00" size="1" pos="92" show="00" value="00"/>
    <field name="smb.andxoffset" showname="AndXOffset: 156" size="2" pos="93" show="156" value="9c00"/>
    <field name="smb.connect.flags" size="2" pos="95" value="0c00">
    <field name="smb.pwlen" showname="Password Length: 1" size="2" pos="97" show="1" value="0100"/>
    <field name="smb.bcc" showname="Byte Count (BCC): 113" size="2" pos="99" show="113" value="7100"/>
    <field name="smb.password" showname="Password: 00" size="1" pos="101" show="00" value="00"/>
    <field name="smb.path" showname="Path: \\SP-SIN-DCF-01.SOUTHPACIFIC.CORP.MICROSOFT.COM\IPC$"
size="106" pos="102" show="\\\\\\SP-SIN-DCF-01.SOUTHPACIFIC.CORP.MICROSOFT.COM\IPC$" value="5c005c00...."/>
    <field name="smb.service" showname="Service: ??????" size="6" pos="208" show="?????" value="3f3f3f3f00"/>
  </field>
</proto>

```

Figure 2: Ethereum’s XML output of an example SMB “Tree Connect AndX Request” message (edited for better presentation).

current un-optimized implementation takes about 6-12 hours for a trace of several million messages (the merging procedure is the slowest due to the need of pairwise comparisons of all inferred formats). Before discussing the experimental results, we first describe our data sets and evaluation methodology.

## 4.1 Data Sets

We tested Discoverer on traces from two different sites: a honeyfarm site [2] (which responds to unsolicited, mostly malicious traffic) and a busy enterprise (which has diverse and high-volume traffic). The honeyfarm trace consists of CIFS/SMB only. The enterprise trace includes HTTP, CIFS/SMB, and RPC. The honeyfarm trace and the HTTP trace were used as the *calibration* data to help guide the design process and set tunable parameters. Our results are presented based on the output of our tool on the traces from the enterprise site, which served as the *evaluation data*. Thus, CIFS/SMB can be seen as the evaluation case where the tool was trained on the trace from a different site, whereas RPC is the case

where the tool is evaluated over a new protocol. Though CIFS/SMB messages may encapsulate the RPC layer, the RPC trace consists of RPC traffic exclusively. The HTTP trace was used for both calibration and evaluation, but we hardly tailored our tool to HTTP.

In our experiment, the CIFS/SMB and RPC trace from the enterprise site contains traffic in one direction only. This will not affect our evaluation because the protocol formats in both directions are equally complicated based on Ethereum’s parsing results of the honeyfarm CIFS/SMB trace. This is not to say that if we can infer the format in one direction, we are guaranteed to infer the format in the other direction; but the performance in one direction does give an indication of the performance in the other direction. In addition, since we do not put messages in different directions into the same cluster, unidirectional traffic does not make the problem any easier.

For the HTTP trace, our tool reassembled consecutive data sent in one direction into a message. For the CIFS/SMB and RPC traces, we leveraged Ethereum to parse them and identify message boundaries. A summary of these traces is shown in Table 2.

```
nbss.type;Length;Server Component;smb.cmd;smb.nt_status;smb.flags;smb.flags2;smb.pid.high;
smb.signature;smb.reserved; smb.tid;smb.pid;smb.uid;smb.mid;smb.wct;AndXCommand;smb.reserved;
smb.andxoffset;smb.connect.flags;smb.pwlen;smb.bcc;smb.password;smb.path;smb.service
```

Figure 3: The “name” of the true format for the example message in Figure 2 concatenates the human readable names of all the fields.

Token	True Field	Token	True Field	Token	True Field	Token	True Field
C(b,00)	nbss.type	C(b,00)	smb.pid.high	C(b,00)	smb.tid	C(b,00)	
C(b,00)	Length	C(b,00)		C(b,00)		V(b,2)	smb.connect.flags
C(b,00)		V(b,256)	smb.signature	C(b,ff)	smb.pid	C(b,00)	
L(b)		V(b,256)		C(b,fe)		C((b,01)	smb.pwlen
C(b,ff)	Server Component	V(b,256)		V(b,33)	smb.uid	C(b,00)	
C(tn,SMBu)	smb.cmd	V(b,256)		V(b,32)		L(b)	smb.bcc
C(b,00)	smb.nt_status	V(b,256)		V(b,13)	smb.mid	C(00)	
C(b,00)		V(b,256)		V(b,211)		C(00)	smb.password
C(b,00)		V(b,256)		C(b,04)	smb.wct	V(tun,664)	smb.path
C(b,18)	smb.flags	V(b,256)		C(b,ff)	AndXCommand	C(tn,????)	smb.service
C(b,07)	smb.flags2	C(b,00)	smb.reserved	C(b,00)	smb.reserved		
C(b,c8)		C(b,00)		L(b)	smb.andxoffset		

Table 1: Discoverer’s inferred format for the true format in Figure 3. For C(x,y), C means constant, x means binary (“b”) or text (“t”; in text tokens, “u” means Unicode and “n” means it is null terminated), y is the hex value or string of the token; for V(x,z), z is the number of different values for the token.

Protocol	Source	Size (B)	# Messages	# True Formats
HTTP	Enterprise	4.6G	5,950,453	2,696
RPC	Enterprise	179M	351,818	50
CIFS/SMB	Enterprise	1.0G	3,818,267	301
CIFS/SMB	Honeyfarm	1.1G	1,439,744	1220

Table 2: Summary of network traces used in the evaluation.

## 4.2 Evaluation Methodology

Our evaluation methodology is to compare the quality of our output with the set of *true* message formats. To obtain the true format, instead of trying to manually extract it from documentation and RFCs, we used the protocol analyzers in Ethereum [5]. Ethereum can parse a network trace and produce, for each message in the trace, an XML output that includes the list of fields in the message, the values of those fields, some human readable names and their sizes. Based on this output, we assign every message a true format “name”, which is simply the concatenation of the human readable names of all the fields. An example of Ethereum’s XML output and the true format name is shown in Figure 2 and Figure 3.

We characterize the performance of our tool and highlight the results in the following metrics:

- *Correctness*: If a cluster contains messages from more than one true format, then Discoverer will

make incorrect inference. Thus we measure the correctness by checking the number of different true formats followed by the messages in a cluster. For all three protocols, over 90% clusters contain messages from a single true format.

- *Conciseness*: Our conservative clustering may cause multiple inferred formats to cover subsets of a single true format. A large number of redundant formats will affect the conciseness of the protocol specifications generated by our tool. Thus we measure conciseness by the ratio from the number of inferred formats to the number of true formats followed by their messages. For all three protocols, we achieved a low 5 to 1 ratio.
- *Coverage*: We measure the trace coverage from two perspectives: the fraction of messages covered by our inferred formats and the fraction of true formats

Parameter	Value
Maximum message prefix	2048 bytes
Minimum length of text segments	3 letters
Minimum cluster size	20 messages
Maximum distinct values for FD	10
Alignment match score	1
Alignment mismatch score	0
Alignment gap score	-2

Table 3: Summary of parameters.

followed by covered messages. For all the three protocols, our message coverage is above 95% while our format coverage is around 30-40%.

As for the semantic inference, all the length fields inferred by Discoverer are correct; certain length fields are missed due to the trace limitation. For instance, some true formats in CIFS/SMB have a fixed message size. In this case, Discoverer will treat the length fields that reflect the message size as constant tokens, and it will not affect parsing messages of these formats in practice.

### 4.3 Tunable Parameters

Discoverer has just a few tunable parameters (see Table 3). For a message larger than 2048 bytes, we only consider the first 2048 bytes, referred to as the maximum message prefix. The minimum length of text segments controls the tokenization procedure (Section 3.2.1). The minimum cluster size and the maximum distinct values for FD are used in the recursive clustering phase (see Section 3.3.3). The match/mismatch/gap scores are parameters for sequence alignment [15]. We observed that the performance of Discoverer is not sensitive to the settings of these parameters. For instance, we saw similar performance when we changed the maximum prefix size from 2048 bytes to 1024 bytes or changed the minimum cluster size from 20 messages to 10 messages. In addition, our type-based sequence alignment is not sensitive to the match/mismatch/gap scores as we discussed in Section 3.4. Thus we take the same parameters for our evaluations on all three protocols.

In the rest of this section, we present the experimental results on the enterprise traces for HTTP, RPC, and CIFS/SMB. Note that we use the inferred format and cluster interchangeably because we infer one format from each cluster.

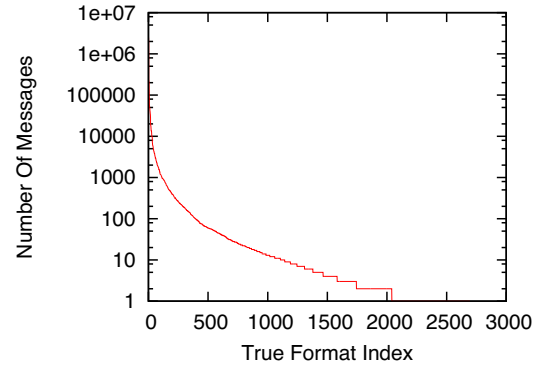


Figure 4: Heavy-tail distribution of message format popularity in HTTP.

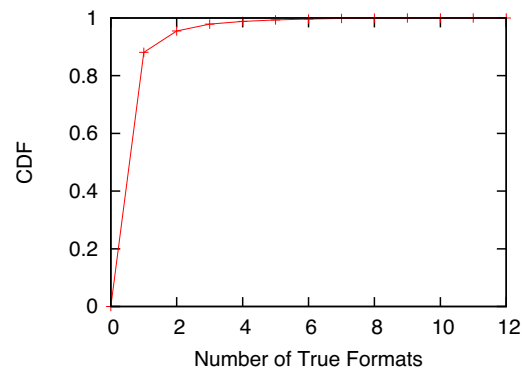


Figure 5: Correctness for HTTP: CDF of the number of true formats followed by messages of a cluster for all clusters before the merging phase.

### 4.4 HTTP

The HTTP protocol allows an arbitrary number of “parameter: value” pairs in an arbitrary order. We refer this to be the “set” semantic. Currently we are unable to identify this set semantic. So we treat each ordering of the set elements as a distinct format. By doing so, we observed 2,696 formats from the parsing results of Ethereum. We leave the identification of set semantic to be future work (see Section 6).

In Figure 4, we show the number of messages of each true format in the HTTP trace. Note that the y-axis is in logarithmic scale. This clearly reveals the heavy-tail distribution; most messages (more than 99%) fall in the first top 1000 true formats. We observed a similar trend in the RPC and CIFS/SMB trace as well. The implication for our tool is that the format coverage and message

coverage are likely to be very different; the latter will be much higher compared to the former. In HTTP, we inferred 3,926 formats, which covered 5,938,511 out of 5,950,453 messages (99.8%). The covered messages belong to 865 out of 2,696 true formats (32%). Since we have a hard requirement on the minimum size of a cluster, we conjecture that the coverage ratio in terms of true formats will improve when the trace grows and each format has more messages.

Figure 5 plots the CDF of the number of true formats followed by messages of a cluster for all clusters before the merging phase. This reflects the correctness of our tool. This figure shows that about 90% of our inferred clusters are correct. They correspond to only one true format. The number rises to over 95% if we include inferred formats that match two true formats.

By manually inspecting the results, we found that the clustering errors are mainly due to the inaccuracy in Ethereum parsing. For example, in some message formats, Discoverer infers that there is a token that can be either “Connection” or “Proxy-Connection”. Discoverer does not treat it as a FD because both may be followed by the same set of values such as “Close” or “Keep-Alive”. However, Ethereum does not recognize “Proxy-Connection” as a parameter for HTTP, and returns a null string for this field in its parsing result, while it returns “http.connection” for “Connection”. So we will have two true formats for a cluster that contains both “Connection” and “Proxy-Connection”. Thus, our conciseness number may improve if Ethereum has more accurate parsing.

The merging phase reduced 4,465 clusters to 3,926 clusters. Since the covered messages belong to 865 true formats, this gives us a 5 to 1 ratio. In fact, almost 80% true formats are scattered into at most five clusters. On one hand, our conservative strategy eliminated false positives (i.e., wrongly merging two clusters that correspond to two different true formats). On the other hand, it did not help much in merging clusters for HTTP. The reason is as follows. HTTP allows many parameters in the form of “parameter: value”. We treat the “parameter:” and “value” as separate tokens because of the space in between. Since the “value” token for certain parameters such as “PROXY” may be arbitrary strings, it is likely for such “value” tokens in two clusters to not have overlapped values. In this case, we will treat them as a mismatch. If two clusters happen to have more than one such mismatch, we will not merge them based on our conservative policy.

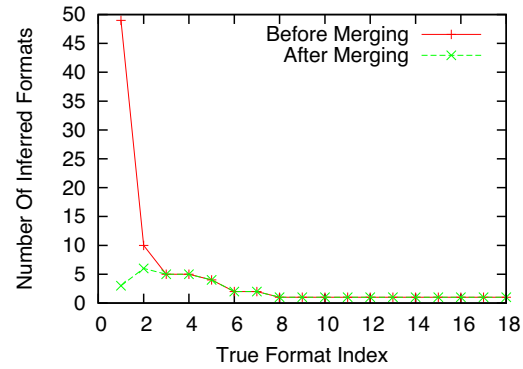


Figure 6: Effectiveness of merging for RPC: Number of inferred formats into which messages of a single true format are “scattered”: before and after merging.

## 4.5 RPC

The RPC trace consists of exclusively RPC traffic. Though the trace size is 179MB, one order less than the HTTP and CIFS/SMB trace, we observed the similar trend in the distribution of the number of messages in each true format. Overall, we inferred 33 formats, which covered 340,624 out of 351,818 messages (96.8%). The covered messages belong to 18 out of 50 true formats (36%).

The recursive clustering generated 83 clusters, among which 78 clusters contain messages from a single true format, and the rest five clusters have messages from two true formats. The merging phase helped reduce the overall clusters from 83 to 33 without introducing false positives. This shows that our merging phase compensates the tokenization errors well by recognizing wrongly classified binary and text tokens. From Figure 6 we can see that, for each of 11 true formats, its messages were merged into in a single cluster.

## 4.6 CIFS/SMB

CIFS/SMB is a fairly complex binary protocol which includes several layers of protocols: it consists of the NetBIOS Session Service (NBSS) headers which encapsulate a SMB header which in turn is layered over RPC. Overall, we inferred 679 formats, which covered 3,640,239 out of 3,818,267 messages (95.3%). The covered messages belong to 130 out of 301 true formats (43%).

In Figure 7, we plot the CDF of the number of true formats followed by messages of a cluster for all clusters before the merging phase. We can see that 57%

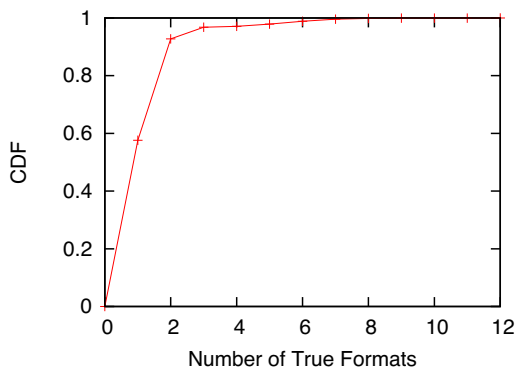


Figure 7: Correctness for CIFS/SMB: CDF of the number of true formats followed by messages of a cluster for all clusters before the merging phase.

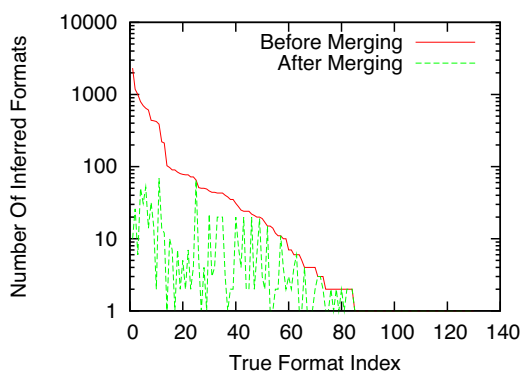


Figure 8: Merging effectiveness for CIFS/SMB: Number of inferred formats into which messages of a single true format are “scattered”: before and after merging

clusters contain messages from a single true format, and 35% clusters have messages from two true formats. We manually checked these clusters and found that it is due to the imprecise parsing of *Ethereal*. It recognized the last field as a “*dcerpc.nt.close.frame*” field for some messages and as a stub data for other messages while those messages have the same format according to our manual inspection. If we take into account this factor, more than 90% clusters contain messages from a single true format, which is consistent with *HTTP*.

We further inspected the clusters consisting of messages from more than two formats and found that, for many of such clusters, the only difference in the true formats followed by their messages is the last field. It is in the form of “Stub data (XX bytes)”, and the difference is in “XX” which says the size of the stub data. Based on

our manual inspection, we conjecture that these stub data likely follow the same format and the size difference is due to a text field with a variable length embedded in the stub data. Therefore, 90% is a conservative measure on the correctness.

In Figure 8, we plot the number of inferred formats into which messages of a single true format are scattered before and after merging. Like *RPC*, the merging technique is effective on *CIFS/SMB*. Overall, we reduced the number of clusters from 7180 to 679 without introducing false positives, which gives us a 5 to 1 ratio against the 130 true formats.

## 5 Related Work

We divide related work into three categories. First, we discuss the state of the art in protocol reverse engineering. Second, we present the previous work that was geared towards a specific application rather than performing all-purpose protocol reverse engineering. Finally, we discuss grammar inference.

To date, most protocol reverse engineering appears to be a painstaking manual process, which involves looking at available documentation, source code, and traces. Two popular examples in the community include the *SAMBA* project [18] and the messenger clients [6]. Automatic protocol reverse engineering appears to have received much less attention. The most closely related work to our paper that we are aware of is the *Protocol Informatics* project [17]. It aims to employ sequence alignment techniques to infer protocol formats from a trace of the protocol. Its main limitation is that the byte-wise sequence alignment, while ideally suited to aligning messages with similar *byte sequences*, is not suitable for aligning messages with similar *formats*. In addition, selecting weights to tune the alignment is hard as shown in [3].

Previous studies have also performed some level of protocol reverse engineering with a specific purpose in mind, namely, application-level replay and protocol identification.

*RolePlayer* [3] and *ScriptGen* [10, 11] both leverage byte-wise sequence alignment techniques to achieve application-level replay by heuristically detecting and adjusting some specific fields such as network addresses, lengths, and cookies. A driving application for application-level replay is to build a protocol-independent, application-level proxy to filter known attacks in a honeyfarm. To improve the performance of the Needleman-Wunsch algorithm [15], *RolePlayer* uses the so-called *pairwise constraint matrix*, which specifies whether the *i*th byte of the first message can or cannot be aligned with the *j*th byte of the second message based on

the field semantic of the  $i$ th byte. However, if the semantic of the  $i$ th byte in the first message is unknown, it can be aligned with any byte in the second message, which may lead to alignment errors. There are two key differences between Discoverer and these two systems. First, RolePlayer and ScriptGen only discover the protocol format to the extent necessary for replay, while Discoverer is aimed to discover the complete protocol format. Second, instead of using the byte-wise sequence alignment, we first cluster messages based on token patterns and then use a novel type-based sequence alignment technique to align and compare message formats based on token types. This represents a significant improvement: on one hand, we can avoid byte-pattern alignment in the recursive clustering phase to achieve a good performance on correctness; on the other hand, we can mitigate overclassification by merging similar inferred formats. Furthermore, compared with ScriptGen which clusters messages by comparing the whole messages at once, our *recursive* clustering technique performs better because we not only look at the potential FD token itself but also look into “the future” by comparing the subsequent tokens in the messages. Some of our techniques for identifying semantically important fields (such as length fields) are borrowed from RolePlayer.

Ma *et al.* [13] perform protocol identification, that is, they classify the set of sessions in a trace into various protocols without relying on port numbers. They develop three techniques for profiling messages exchanged in a protocol: product distributions of byte offsets, Markov models of byte transitions, and common substring graphs of message strings. The main difference between their work and ours is that we have different goals. They aim to characterize a protocol based on the first  $n$  (e.g., 64) bytes in sessions of the protocol; we leverage the format inference and type-based sequence alignment techniques to decipher the message formats of the entire session.

The problem of protocol reverse engineering is related to the theoretical problem of grammar inference, which aims to deduce the grammar given a set of sample strings drawn from it. This problem is unfortunately theoretically unsolvable, even when the grammar is in the simplest form of Chomskian grammar, the regular language [7]. Since even a regular language can be potentially infinite and the sample set cannot be, it turns out that this task is impossible. The language implicit in application-level protocols is often substantially more complex than a regular language, involving fields such as length fields. Because of this complexity, we were unable to directly apply any results from the grammar inference community. There have been extensions based on Kolmogorov complexity [4] to learn the simplest finite

language from only positive examples, but once again, they appear too complicated to apply to the context sensitive grammars that network protocols involve.

Techniques used in the speech recognition community, such as, probabilistic Markov chain analysis [8], were not applied in our work, since the correlation between protocol fields makes it difficult for the byte sequence in a message to be modeled as independent samples from a Markov process.

## 6 Limitations and Future Work

In this section, we discuss the limitations of our approach. We categorize our limitations into two categories: ones that are fundamental to the problem we want to solve and those that are due to the heuristics in our tool. We will also describe future research directions for solving these limitations.

There are two main fundamental limitations.

- *Trace Dependency*: The format generated by any tool that operates only on the trace is limited by the diversity of traffic seen in the trace. If certain message formats never occur in the trace, or if certain variable fields never take more than one value in the trace, it is impossible for such a tool to infer those message formats or identify those fields as variable fields.
- *Pre-Defined Semantics*: Only a set of pre-defined semantics can be inferred. Since it is not possible to find all the possible semantics of all fields just from a trace, the best one can hope for is to have an extensible framework where new semantic modules can be added as desired.

We now move on to the imprecision problems that are directly related to the design of our tool. The following are the major imprecisions in our inferred message formats:

- *Semantics*: At present, we cannot capture the following semantics. (a) Set semantics: For instance, HTTP allows an arbitrary number of parameters to be specified in any order. Identifying this list of supported parameters as a set that allows re-ordering during encoding would considerably improve our performance. (b) Pointer field: This is a field whose value is the offset of another field in an array of some arbitrary items. Such fields occur in DNS. (c) Array length: This is a field whose value is the number of items in an array of some arbitrary items (e.g., DNS). We plan to study the inference of these semantics in the future.

- *Coalescing Fields*: We identify a binary field as a sequence of binary tokens each spanning a single byte. This is a limitation since ideally we would want such a field identified as a single binary token. Unlike text fields, no clue may be available in delimiting binary fields. The only way out is techniques based on frequency analysis (e.g., does this byte vary as much as the other one?). However, this kind of techniques tend to be unreliable. For instance, in a two-byte process ID field, the more significant byte may change much less frequently than the less significant one since an operating system usually issues process IDs incrementally from zero. Thus we chose to list such fields as a series of single byte tokens. Our plan is to enrich our semantic inference modules so that a sequence of binary bytes with a common semantic can be identified as a single field. For example, a length field spanning four bytes will be identified as a single field because of the semantic module for detecting length fields.
- *Asynchronous Protocols*: With asynchronous protocols, it is difficult to even delimit messages from network packets. This is because messages in one direction may be interrupted by those in the other direction, and messages in one direction may be delayed allowing two back-to-back messages in the other direction. We have not experimented with any asynchronous protocols so far.
- *Application Sessions*: Currently, our tool analyzes each connection in isolation. However, if we had a good session description of the various connections and various hosts involved, it would be trivial to process the trace with such session knowledge, and derive formats for the whole session. A previous study [9] aimed to semi-automatically discover session structures.
- *State Machine Inference*: Currently, we only envision a state machine constructed from the trace by using the inferred message formats to assign a type to each message, and then simply inferring the FSA that captures the sequences of messages in all sessions in the trace. However, this is hardly the compact FSA that the application developer had in mind. In such case, using FSA minimization techniques [19] may simplify the FSA considerably.

Many of the limitations above are due to the limited information available from network traces. To tackle these limitations and achieve a better reverse-engineered protocol specification, we can use program analysis to gain

more information and insight into the parsing and processing of the input in the program. For instance, we may easily identify two consecutive bytes as a WORD (i.e., a two-byte integer) from run-time analysis by observing that they are processed as a WORD throughout the execution.

We have focused on reverse engineering network protocols in Discoverer; it would be useful to reverse engineer the input specifications for file-based applications, since we have seen a significant growth in file-based attacks.

## 7 Conclusion

Protocol reverse engineering is a highly manual process today, which is still suffered through because of the immense value of protocol knowledge. We have developed Discoverer, a tool that aims to automate this reverse engineering process. Discoverer leverages recursive clustering and type-based sequence alignment to infer message formats. We have demonstrated Discoverer can infer message formats effectively for three network protocols, CIFS/SMB, RPC, and HTTP. In the future, we plan to enrich our semantic inference, research on the protocol state machine inference, explore the direction of using program analysis to reverse engineer the specifications of both network and file input, and apply reverse-engineered protocol specifications to real world applications.

## Acknowledgments

We would like to thank Vern Paxson, Ion Stoica, Christian Kreibich, and Gautam Altekar for their valuable comments on an early draft of this paper. We thank Joseph Kravis for providing network traces to us. We would also like to thank the anonymous reviewers for their insightful comments.

## References

- [1] N. Borisov, D. J. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. A Generic Application-Level Protocol Analyzer and its Language. In *Proceedings of the 14th Annual Network & Distributed System Security Symposium (NDSS)*, March 2007.
- [2] W. Cui, V. Paxson, and N. Weaver. GQ: Realizing a System to Catch Worms in a Quarter Million Places. Technical Report TR-06-004, ICSI, 2006.
- [3] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-Independent Adaptive Replay of Application

- Dialog. In *Proceedings of the 13<sup>th</sup> Symposium on Network and Distributed System Security (NDSS 2006)*, February 2006.
- [4] F. Denis. Learning Regular Languages from Simple Positive Examples. *Machine Learning*, 44(1/2):37–66, 2001.
- [5] Ethereum: A Network Protocol Analyzer. <http://ethereum.com>.
- [6] Gaim Instant Messaging Client. <http://gaim.sourceforge.net>.
- [7] E. M. Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, 1967.
- [8] F. Jelinek and J. D. Lafferty. Computation of the Probability of Initial Substring Generation by Stochastic Context-Free Grammars. *Computational Linguistics*, 17(3):315–323, 1991.
- [9] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal. Semi-Automated Discovery of Application Session Structure. In *Proceedings of the 2006 Internet Measurement Conference (IMC)*, Rio de Janeiro, Brazil, October 2006.
- [10] C. Leita, M. Dacier, and F. Massicotte. Automatic Handling of Protocol Dependencies and Reaction to 0-Day Attacks with ScriptGen based Honeyd. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection*, Hamburg, Germany, September 2006.
- [11] C. Leita, K. Mermoud, and M. Dacier. ScriptGen: An Automated Script Generation Tool for Honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005)*, December 2005.
- [12] The libpcap Project. <http://sourceforge.net/projects/libpcap/>.
- [13] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. Voelker. Unexpected Means of Protocol Inference. In *Proceedings of the 2006 Internet Measurement Conference*, 2006.
- [14] Microsoft Corporation. Microsoft Network Monitor 3. <http://www.microsoft.com/downloads/details.aspx?FamilyID=aa8be06d-4a6a-4b69-b861-2043b665cb53>.
- [15] S. B. Needleman and C. D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.
- [16] R. Pang, V. Paxson, R. Somer, and L. Peterson. binpac: A yacc for Writing Application Protocol Parsers. In *Proceedings of the 2006 Internet Measurement Conference*, October 2006.
- [17] The Protocol Informatics Project. <http://www.baselineresearch.net/PI/>.
- [18] How Samba Was Written. <http://samba.org/ftp/tridge/misc/french-cafe.txt>.
- [19] B. W. Watson. A Taxonomy of Finite Automaton Minimization Algorithms. Technical Report CS-93-44, Eindhoven University of Technology, Eindhoven, The Netherlands, 1993.

# Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation

David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, Dawn Song  
Carnegie Mellon University  
{dbrumley,jcaballero,zliang,jnewsome,dawnsong}@cmu.edu

## Abstract

Different implementations of the same protocol specification usually contain *deviations*, i.e., differences in how they check and process some of their inputs. Deviations are commonly introduced as implementation errors or as different interpretations of the same specification. Automatic discovery of these deviations is important for several applications. In this paper, we focus on automatic discovery of deviations for two particular applications: error detection and fingerprint generation.

We propose a novel approach for automatically detecting deviations in the way different implementations of the same specification check and process their input. Our approach has several advantages: (1) by automatically building symbolic formulas from the implementation, our approach is precisely faithful to the implementation; (2) by solving formulas created from two different implementations of the same specification, our approach significantly reduces the number of inputs needed to find deviations; (3) our approach works on binaries directly, without access to the source code.

We have built a prototype implementation of our approach and have evaluated it using multiple implementations of two different protocols: HTTP and NTP. Our results show that our approach successfully finds deviations between different implementations, including errors in input checking, and differences in the interpretation of the specification, which can be used as fingerprints.

## 1 Introduction

Many different implementations usually exist for the same protocol. Due to the abundance of coding errors and protocol specification ambiguities, these implementations usually contain *deviations*, i.e., differences in how they check and process some of their inputs. As a result, same inputs can cause different implementations to reach

semantically different protocol states. For example, an implementation may not perform sufficient input checking to verify if an input is well-formed as specified in the protocol specification. Thus, for some inputs, it might exhibit a deviation from another implementation, which follows the protocol specification and performs the correct input checking.

Finding these deviations in implementations is important for several applications. In particular, in this paper we show 1) how we can automatically discover these deviations, and 2) how we can apply the discovered deviations to two particular applications: *error detection* and *fingerprint generation*.

First, finding a deviation between two different implementations of the same specification may indicate that at least one of the two implementations has an error, which we call *error detection*. Finding such errors is important to guarantee that the protocol is correctly implemented, to ensure proper interoperability with other implementations, and to enhance system security since errors often represent vulnerabilities that can be exploited. Enabling error detection by automatically finding deviations between two different implementations is particularly attractive because it does not require a manually written model of the protocol specification. These models are usually complex, tedious, and error-prone to generate. Note that such deviations do not necessarily flag an error in one of the two implementations, since deviations can also be caused by ambiguity in the specification or when some parts are not fully specified. However, automatic discovery of such deviations is a good way to provide candidate implementation errors.

Second, such deviations naturally give rise to *fingerprints*, which are inputs that, when given to two different implementations, will result in semantically different output states. Fingerprints can be used to distinguish between the different implementations and we call the discovery of such inputs *fingerprint generation*. Fingerprinting has been in use for more than a decade [25]

and is an important tool in network security for remotely identifying which implementation of an application or operating system a remote host is running. Fingerprinting tools [8, 11, 15] need fingerprints to operate and constantly require new fingerprints as new implementations, or new versions of existing implementations, become available. Thus, the process of automatically finding these fingerprints, i.e., the fingerprint generation, is crucial for these tools.

Automatic deviation discovery is a challenging task—deviations usually happen in corner cases, and discovering deviations is often like finding needles in a haystack. Previous work in related areas is largely insufficient. For example, the most commonly used technique is random or semi-random generation of inputs [20, 43] (also called fuzz testing). In this line of approach, random inputs are generated and sent to different implementations to observe if they trigger a difference in outputs. The obvious drawback of this approach is that it may take many such random inputs before finding a deviation.

In this paper, we propose a novel approach to automatically discover deviations in input checking and processing between different implementations of the same protocol specification. We are given two programs  $P_1$  and  $P_2$  implementing the same protocol. At a high level, we build two formulas,  $f_1$  and  $f_2$ , which capture how each program processes a single input. Then, we check whether the formula  $(f_1 \wedge \neg f_2) \vee (\neg f_1 \wedge f_2)$  is satisfiable, using a solver such as a decision procedure. If the formula is satisfiable, it means that we can find an input, which will satisfy  $f_1$  but not  $f_2$  or vice versa, in which case it may lead the two program executions to semantically different output states. Such inputs are good candidates to trigger a deviation. We then send such candidate inputs to the two programs and monitor their output states. If the two programs end up in two semantically different output states, then we have successfully found a deviation between the two implementations, and the corresponding input that triggers the deviation.

We have built a prototype implementation of our approach. It handles both Windows and Linux binaries running on an x86 platform. We have evaluated our approach using multiple implementations of two different protocols: HTTP and NTP. Our approach has successfully identified deviations between servers and automatically generated inputs that triggered different server behaviors. These deviations include errors and differences in the interpretation of the protocol specification. The evaluation shows that our approach is accurate: in one case, the relevant part of the generated input is only three bits. Our approach is also efficient: we found deviations using a single request in about one minute.

**Contributions.** In summary, in this paper, we make the following contributions:

- **Automatic discovery of deviations:** We propose a novel approach to automatically discover deviations in the way different implementations of the same protocol specification check and process their input. Our approach has several advantages: (1) by automatically building symbolic formulas from an implementation, our approach is precisely faithful to the implementation; (2) by solving formulas created from two different implementations of the same specification, our approach significantly reduces the number of inputs needed to find deviations; (3) our approach works on binaries directly, without access to the source code. This is important for wide applicability, since implementations may be proprietary and thus not have the source code available. In addition, the binary is what gets executed, and thus it represents the true behavior of the program.
- **Error detection using deviation discovery:** We show how to apply our approach for automatically discovering deviations to the problem of error detection—the discovered deviations provide candidate implementation errors. One fundamental advantage of our approach is that it does not require a user to manually generate a model of the protocol specification, which is often complex, tedious, and error-prone to generate.
- **Fingerprint generation using deviation discovery:** We show how to apply our approach for automatically discovering deviations to the problem of fingerprint generation—the discovered deviations naturally give rise to fingerprints. Compared to previous approaches, our solution significantly reduces the number of candidate inputs that need to be tested to discover a fingerprint [20].
- **Implementing the approach:** We have built a prototype that implements our approach. Our evaluation shows that our approach is accurate and efficient. It can identify deviations with few example inputs at bit-level accuracy.

The remainder of the paper is organized as follows. Section 2 introduces the problem and presents an overview of our approach. In Section 3 we present the different phases and elements that comprise our approach and in Section 4 we describe the details of our implementation. Then, in Section 5 we present the evaluation results of our approach over different protocols. We discuss future enhancements to our approach in Section 6. Finally, we present the related work in Section 7 and conclude in Section 8.

## 2 Problem Statement and Approach Overview

In this section, we first describe the problem statement, then we present the intuition behind our approach, and finally we give an overview of our approach.

**Problem statement.** In this paper we focus on the problem of automatically detecting deviations in protocol implementations. In particular, we aim to find inputs that cause two different implementations of the same protocol specification to reach semantically different output states. When we find such an input, we say we have found a candidate deviation.

The output states need to be externally observable. We use two methods to observe such states: (a) monitoring the network output of the program, and (b) supervising its environment, which allows us to detect unexpected states such as program halt, reboot, crash, or resource starvation. However, we cannot simply compare the complete output from both implementations, since the output may be different but semantically equivalent. For example, many protocols contain sequence numbers, and we would expect the output from two different implementations to contain two different sequence numbers. However, the output messages may still be semantically equivalent.

Therefore, we may use some domain knowledge about the specific protocol being analyzed to determine when two output states are semantically different. For example, many protocols such as HTTP, include a status code in the response to provide feedback about the status of the request. We use this information to determine if two output states are semantically equivalent or not. In other cases, we observe the effect of a particular query in the program, such as program crash or reboot. Clearly these cases are semantically different from a response being emitted by the program.

**Intuition of our approach.** We are given two implementations  $P_1$  and  $P_2$  of the same protocol specification. Each implementation at a high level can be viewed as a mapping function from the protocol input space  $I$  to the protocol output state space  $S$ . Let  $P_1, P_2 : I \rightarrow S$  represent the mapping function of the two implementations. Each implementation accepts inputs  $x \in I$  (e.g., an HTTP request), and then processes the input resulting in a particular protocol output state  $s \in S$  (e.g., an HTTP reply). At a high level, we wish to find inputs such that the same input, when sent to the two implementations, will cause each implementation to result in a different protocol output state.

Our goal is to find an input  $x \in I$  such that  $P_1(x) \neq P_2(x)$ . Finding such an input through random testing is usually hard.

However, in general it is easy to find an input  $x \in I$  such that  $P_1(x) = P_2(x) = s \in S$ , i.e., most inputs will result in the same protocol output state  $s$  for different implementations of the same specification. Let  $f(x)$  be the formula representing the set of inputs  $x$  such that  $f(x) = \text{true} \iff P(x) = s$ . When  $P_1$  and  $P_2$  implement the same protocol differently, there may be some input where  $f_1$  will not be the same as  $f_2$ :

$$\exists x. (f_1(x) \wedge \neg f_2(x)) \vee (\neg f_1(x) \wedge f_2(x)) = \text{true}.$$

The intuition behind the above expression is that when  $f_1(x) \wedge \neg f_2(x) = \text{true}$ , then  $P_1(x) = s$  (because  $f_1(x) = \text{true}$ ) while  $P_2(x) \neq s$  (because  $f_2(x) = \text{false}$ ), thus the two implementations reach different output states for the same input  $x$ . Similarly,  $\neg f_1(x) \wedge f_2(x)$  indicates when  $P_1(x) \neq s$ , but  $P_2(x) = s$ . We take the disjunction since we only care whether the implementations differ from each other.

Given the above intuition, the central idea is to create the formula  $f$  using the technique of weakest precondition [19, 26]. Let  $Q$  be a predicate over the state space of a program. The weakest precondition  $wp(P, Q)$  for a program  $P$  and post-condition  $Q$  is a boolean formula  $f$  over the input space of the program. In our setting, if  $f(x) = \text{true}$ , then  $P(x)$  will terminate in a state satisfying  $Q$ , and if  $f(x) = \text{false}$ , then  $P(x)$  will not terminate in a state satisfying  $Q$  (it either “goes wrong” or does not terminate). For example, if the post-condition  $Q$  is that  $P$  outputs a successful HTTP reply, then  $f = wp(P, Q)$  characterizes all inputs which lead  $P$  to output a successful HTTP reply. The boolean formula output by the weakest precondition is our formula  $f$ .

Furthermore, we observe that the above method can still be used even if we do not consider the entire program and only consider a *single* execution path (we discuss multiple execution paths in Section 6). In that case, the formula  $f$  represents the subset of protocol inputs that follow one of the execution paths considered and still reach the protocol output state  $s$ . Thus,  $f(x) = \text{true} \Rightarrow P(x) = s$ , since if an input satisfies  $f$  then for sure it will make program  $P$  go to state  $s$ , but the converse is not necessarily true—an input which makes  $P$  go to state  $s$  may not satisfy  $f$ . In our problem, this means that the difference between  $f_1$  and  $f_2$  may not necessarily result in a true deviation, as shown in Figure 2. Instead, the difference between  $f_1$  and  $f_2$  is a good candidate, which we can then test to validate whether it is a true deviation.

**Overview of our approach.** Our approach is an iterative process, and each iteration consists of three phases, as shown in Figure 1. First, in the *formula extraction* phase, we are given two binaries  $P_1$  and  $P_2$  implementing the same protocol specification, such as HTTP, and

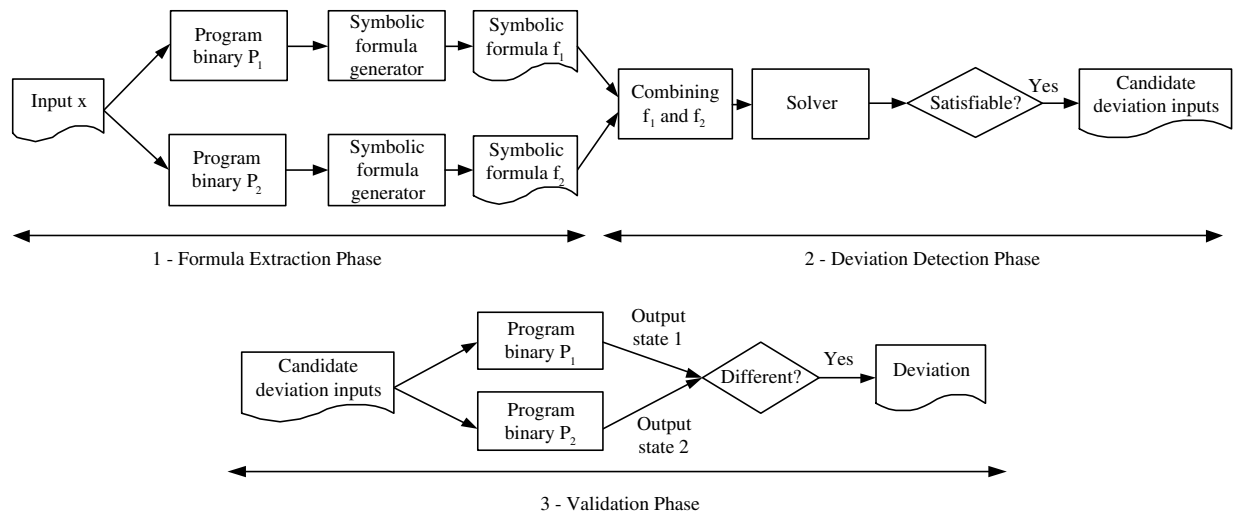


Figure 1: Overview of our approach.

an input  $x$ , such as an HTTP GET request. For each implementation, we log an execution trace of the binary as it processes the input, and record what output state it reaches, such as halting or sending a reply. We assume that the execution from both binaries reaches semantically equivalent output states; otherwise we have already found a deviation! For each implementation  $P_1$  and  $P_2$ , we then use this information to produce a boolean formula over the input,  $f_1$  and  $f_2$  respectively, each of which is satisfied for inputs that cause the binary to reach the same output state as the original input did.

Next, in the *deviation detection* phase, we use a solver (such as a decision procedure) to find differences in the two formulas  $f_1$  and  $f_2$ . In particular, we ask the solver if  $(f_1 \wedge \neg f_2) \vee (f_2 \wedge \neg f_1)$  is satisfiable. When satisfiable the solver will return an example satisfying input. We call these inputs the *candidate deviation inputs*.

Finally, in the *validation* phase we evaluate the candidate deviation inputs obtained in the formula extraction phase on both implementations and check whether the implementations do in fact reach semantically different output states. This phase is necessary because the symbolic formula might not include all possible execution paths, then an input that satisfies  $f_1$  is guaranteed to make  $P_1$  reach the same semantically equivalent output state as the original input  $x$  but an input that does not satisfy  $f_1$  may also make  $P_1$  reach a semantically equivalent output state. Hence, the generated candidate deviation inputs may actually still cause both implementations to reach semantically equivalent output states.

If the implementations *do* reach semantically different output states, then we have found a deviation triggered by that input. This deviation is useful for two things: (1) it

may represent an implementation error in at least one of the implementations, which can then be checked against the protocol specification to verify whether it is truly an error; (2) it can be used as a *fingerprint* to distinguish between the two implementations.

**Iteration.** We can iterate this entire process to examine *other* input types. Continuing with the HTTP example, we can compare how the two implementations process other types of HTTP requests, such as HEAD and POST, by repeating the process on those types of requests.

## 3 Design

In this section, we describe the details of the three phases in our approach, the formula extraction phase, the deviation detection phase, and the validation phase.

### 3.1 Formula Extraction Phase

#### 3.1.1 Intuition and Overview

The goal of the formula extraction phase is that given an input  $x$  such that  $P_1(x) = P_2(x) = s$ , where  $s$  is the output state when executing input  $x$  with the two given programs, we would like to compute two formulas,  $f_1$  and  $f_2$ , such that,

$$f_1(x) = \text{true} \Rightarrow P_1(x) = s$$

and

$$f_2(x) = \text{true} \Rightarrow P_2(x) = s,$$

This matches well with the technique of *weakest precondition* (WP) [19, 26]. The weakest precondition, denoted

$wp(P, Q)$ , is a boolean formula  $f$  over the input space  $I$  of  $P$  such that if  $f(x) = \text{true}$ , then  $P(x)$  will terminate in a state satisfying  $Q$ . In our setting, the post-condition is the protocol output state, and the weakest precondition is a formula characterizing protocol inputs, which will cause the implementation to reach the specified protocol output state.

Unfortunately, calculating the weakest precondition over an entire real-world binary program can easily result in a formula that is too big to solve. First, there may be many program paths which can lead to a particular output state. We show that we can generate interesting deviations even when considering a single program path. Second, we observe that in many cases only a small subset of instructions operate on data derived from the original input. There is no need to model the instructions that do not operate on data derived from the original input, since the result they compute will be the same as in the original execution. Therefore we eliminate these instructions from the WP calculation, and replace them with only a series of assignments of concrete values to the relevant program state just before an instruction operates on data derived from the input.

Hence, in our design, we build the symbolic formula in two distinct steps. We first execute the program on the original input, while recording a trace of the execution. We then use this execution trace to build the symbolic formula.

### 3.1.2 Calculating the Symbolic Formula

In order to generate the symbolic formula, we perform the following steps:

1. Record the execution trace of the executed program path.
2. Process the execution trace. This step translates the execution trace into a program  $B$  written in our simplified intermediate representation (IR).
3. Generate the appropriate post-condition  $Q$ .
4. Calculate the weakest precondition on  $B$  by:
  - (a) Translating  $B$  into a single assignment form.
  - (b) Translating the (single assignment) IR program into the guarded command language (GCL). The GCL program, denoted  $B_g$ , is semantically equivalent to the input IR statements, but appropriate for the weakest precondition calculation.
  - (c) Computing the weakest precondition  $f = wp(B_g, Q)$  in a syntax-directed fashion on the GCL.

The output of this phase is the symbolic formula  $f$ . Below we describe these steps in more detail.

**Step 1: Recording the execution trace.** We generate formulas based upon the program path for a single execution. We have implemented a path recorder which records the execution trace of the program. The execution trace is the sequence of machine instructions executed, and for each executed instruction, the value of each operand, whether each operand is derived from the input, and if it is derived from the input, an identifier for the original input stream it comes from. The trace also has information about the first use of each input byte, identified by its offset in the input stream. For example, for data derived from network inputs, the identifier specifies which session the input came from, and the offset specifies the original position in the session data.

**Step 2: Processing the execution trace.** We process the execution trace to include only relevant instructions. An instruction is relevant if it operates on data derived from the input  $I$ . For each relevant instruction, we:

- Translate the x86 instruction to an easier-to-analyze intermediate representation (IR). The generated IR is semantically equivalent to the original instruction.

The advantage of our IR is that it allows us to perform subsequent steps over the simpler IR statements, instead of the hundreds of x86 instructions. The translation from an x86 instruction to our IR is designed to correctly model the semantics of the original x86 instruction, including making otherwise implicit side effects explicit. For example, we insert code to correctly model instructions that set the `eflag` register, single instruction loops (e.g., `rep` instructions), and instructions that behave differently depending on the operands (e.g., shifts).

Our IR is shown in Table 1. We translate x86 instruction into this IR. Our IR has assignments ( $r := v$ ), binary and unary operations ( $r := r_1 \square_b v$  and  $r := \square_u v$  where  $\square_b$  and  $\square_u$  are binary and unary operators), loading a value from memory into a register ( $r_1 := *(r_2)$ ), storing a value ( $*(r_1) := r_2$ ), direct jumps (`jmp  $\ell$` ) to a known target label (label  $\ell_i$ ), indirect jumps to a computed value stored in a register (`ijmp  $r$` ), and conditional jumps (`if  $r$  then jmp  $\ell_1$  else jmp  $\ell_2$` ).

- Translate the information logged about the operands into a sequence of initialization statements. For each operand:
  - If it is not derived from input, the operand is assigned the concrete value logged in the execution trace. These assignments effectively model the sequences of instructions that we do not explicitly include.

<i>Instructions</i>	$i$	$::=$	$*(r_1) := r_2 \mid r_1 := *(r_2) \mid r := v \mid r := r_1 \square_b v$ $\mid r := \square_u v \mid \text{label } l_i \mid \text{jmp } \ell \mid \text{ijmp } r$ $\mid \text{if } r \text{ jmp } \ell_1 \text{ el } e \text{ jmp } \ell_2$
<i>Operations</i>	$\square_b$	$::=$	$+, -, *, /, \ll, \gg, \&,  , \oplus, ==, !=, <, \leq$ (Binary operations)
	$\square_u$	$::=$	$\neg, !$ (unary operations)
<i>Operands</i>	$v$	$::=$	$n$ (an integer literal) $\mid r$ (a register) $\mid \ell$ (a label)
<i>Reg. Types</i>	$\tau$	$::=$	$\text{reg64\_t} \mid \text{reg32\_t} \mid \text{reg16\_t} \mid \text{reg8\_t} \mid \text{reg1\_t}$ (number of bits)

Table 1: Our RISC-like assembly IR. We convert x86 assembly instructions into this IR.

- For operands derived from input, the *first* time we encounter a byte derived from a particular input identifier and offset, we initialize the corresponding byte of the operand with a *symbolic* value that uniquely identifies that input identifier and offset. On subsequent instructions that operate on data derived from that particular input identifier and offset, we do *not* initialize the corresponding operand, since we want to accurately model the sequence of computations on the input.

The output of this step is an IR program  $B$  consisting of a sequence of IR statements.

**Step 3: Setting the post-condition.** Once we have generated the IR program from the execution trace, the next step is to select a post-condition, and compute the weakest precondition of this post-condition over the program, yielding our symbolic formula.

The post-condition specifies the desired protocol output state, such as what kind of response to a request message is desired. In our current setting, an ideal post-condition would specify that “The input results in an execution that results in an output state that is semantically equivalent to the output state reached when processing the original input.” That is, we want our formula to be true for exactly the inputs that are considered “semantically equivalent” to the original input by the modeled program binary.

In our approach, the post-condition specified the output state should be the same as in the trace. In order to make the overall formula size reasonable, we add additional constraints to the post-condition which constraint the formula to the same program path taken as in the trace. We do this by iterating over all conditional jumps and indirect jumps in the IR, and for each jump, add a clause to the post-condition that ensures that the final formula only considers inputs that also result in the same destination for the given jump. For example, if in the trace `if  $e$  then  $\ell_1$  el  $e$   $\ell_2$`  was evaluated and the next instruction executed was  $\ell_2$ , then  $e$  must have evaluated to *false*, and we add a clause restricting  $e = \text{false}$  to the post-condition.

In some programs, there may be multiple paths that reach the same output state. Our techniques can be generalized to handle this case, as discussed in Section 6. In practice, we have found this post-condition to be sufficient for finding interesting deviations. Typically, inputs that cause the same execution path to be followed are treated equivalently by the program, and result in equivalent output states. Conversely, inputs that follow a different execution path often result in a semantically different output state of the program. Although more complicated and general post-conditions are possible, one interesting result from our experiments is that the simple approach was all that was needed to generate interesting deviations.

**Step 4: Calculating the weakest precondition.** The weakest precondition (WP) calculation step takes as input the IR program  $B$  from Step 2, and the desired post-condition  $Q$  from Step 3. The weakest precondition, denoted  $wp(B, Q)$ , is a boolean formula  $f$  over the input space such that if  $f(x) = \text{true}$ , then  $B(x)$  will terminate in a state satisfying  $Q$ . For example, if the program is  $B : y = x + 1$  and  $Q : 2 < y < 5$ , then  $wp(B, Q)$  is  $1 < x < 4$ .

We describe the steps for computing the weakest precondition below.

*Step 4a: Translating into single assignment form.* We translate the IR program  $B$  from the previous step into a form in which every variable is assigned at most once. (The transformed program is semantically equivalent to the input IR.) We perform this step to enable additional optimizations described in [19, 29, 36], which further reduce the formula size. For example, this transformation will rewrite the program  $\vdots \vdots$  as  $\vdots \vdots$ . We carry out this transformation by maintaining a mapping from the variable name to its current incarnation, e.g., the original variable  $x$  may have incarnations  $x_1$ ,  $x_2$ , and  $x_3$ . We iterate through the program and replace each variable use with its current incarnation. This step is similar to computing the SSA form of a program [39], and is a widely used technique.

*Step 4b: Translating to GCL.* The translation to GCL takes as input the single assignment form from step 4a,

and outputs a semantically equivalent GCL program  $B_g$ . We perform this step since the weakest precondition is calculated over the GCL language [26]. The resulting program  $B_g$  is semantically equivalent to the input single-assignment IR statements. The weakest precondition is calculated in a syntax-directed manner over  $B_g$ .

The GCL language constructs we use are shown in Table 2. Although GCL may look unimpressive, it is sufficiently expressive for reasoning about complex programs [24, 26, 28, 29]<sup>1</sup>. Statements  $S$  in our GCL programs will mirror statements in assembly, e.g., store, load, assign, etc. GCL has assignments of the form  $lhs := e$  where  $lhs$  is a register or memory location, and  $e$  is a (side-effect) free expression. **assume**  $e$  assumes a particular (side-effect free) expression is true. An **assume** statement is used to reason about conditional jump predicates, i.e., we add “**assume**  $e$ ” for the true branch of a conditional jump, and “**assume**  $\neg e$ ” for the false branch of the conditional jump. **assert**  $e$  asserts that  $e$  must be true for execution to continue, else the program fails. In other words,  $Q$  cannot be satisfied if **assert**  $e$  is false. **skip** is a semantic no-op.  $S_1; S_2$  denotes a sequence where first statement  $S_1$  is executed and then statement  $S_2$  is executed.  $S_1 \square S_2$  is called a choice statement, and indicates that either  $S_1$  or  $S_2$  may be executed. Choice statements are used for if-then-else constructs.

For example, the IR:

```
if ( $x_0 < 0$ ) {
     $x_1 := x_0 - 1$ ;
} else {
     $x_1 := x_0 + 1$ ;
}
```

will be translated as:

$$(\mathbf{assume} \ x_0 < 0; x_1 = x_0 - 1;) \square$$

$$(\mathbf{assume} \ \neg(x_0 < 0); x_1 := x_0 + 1;)$$

The above allows calculating the WP over multiple paths (we discuss multiple paths in Section 6). In our setting, we only consider a single path. For each branch condition  $e$  evaluated in the trace, we could add the GCL statement **assert**  $e$  if  $e$  evaluated to *true* (else **assert**  $\neg e$  if  $e$  evaluated to *false*). In our implementation, using **assert** in this manner is equivalent to adding a clause for each branch predicate to the post-condition (e.g., making the post-condition  $e \wedge Q$  when  $e$  evaluated to *true* in the trace).

*Step 4c: Computing the weakest precondition.* We compute the weakest precondition for  $B_g$  from the previous step in a syntax-directed manner. The rules for computing the weakest precondition are shown in Table 2. Most rules are straightforward, e.g., to calcu-

<sup>1</sup>The GCL defines a few additional commands such as a **do-while** loop, which we do not need.

late the weakest precondition  $wp(A; B, Q)$ , we calculate  $wp(A, wp(B, Q))$ . Similarly  $wp(\mathbf{assume} \ e, Q) \equiv e \Rightarrow Q$ . For assignments  $lhs := e$ , we generate a let expression which binds the variable name  $lhs$  to the expression  $e$ . We also take advantage of a technical transformation, which can further reduce the size of the formula by using the single assignment form from Step 4a [19, 29, 36].

### 3.1.3 Memory Reads and Writes to Symbolic Addresses

If the instruction accesses memory using an address that is derived from the input, then in the formula the address will be symbolic, and we must choose what set of possible addresses to consider. In order to remain sound, we add a clause to our post-condition to only consider executions that would calculate an address within the selected set. Considering more possible addresses increases the generality of our approach, at the cost of more analysis.

**Memory reads.** When reading from a memory location selected by an address derived from the input, we must process the memory locations in the set of addresses being considered as operands, generating any appropriate initialization statements, as above.

We achieve good results considering only the address that was actually used in the logged execution trace and adding the corresponding constraints to the post-condition to preserve soundness. In practice, if useful deviations are not found from the corresponding formula, we could consider a larger range of addresses, achieving a more descriptive formula at the cost of performance. We have implemented an analysis that bounds the range of symbolic memory addresses [2], but have found we get good results without performing this additional step.

**Memory writes.** We need not transform writes to memory locations selected by an address derived from the input. Instead we record the selected set of addresses to consider, and add the corresponding clause to the post-condition to preserve soundness. These conditions force the solver to reason about any potential alias relationships. As part of the weakest precondition calculation, subsequent memory reads that could use one of the addresses being considered are transformed to a conditional statement handling these potential aliasing relationships.

As with memory reads, we achieve good results only considering the address that was actually used in the logged execution trace. Again, we could generalize the formula to consider more values, by selecting a range of addresses to consider.

$A, B \in \text{GCL stmt}$	$::= lhs := e$	GCL stmt	$\text{wp}(\text{stmt}, Q)$
	$A; B$	<b>assume</b> $e$	$e \Rightarrow Q$
	<b>assume</b> $e$ ( $e$ is an expression)	<b>assert</b> $e$	$e \wedge Q$
	<b>assert</b> $e$ ( $e$ is an expression)	$lhs := e$	$\text{let } lhs = e$
	$A \sqcap B$	$A; B$	$\text{wp}(A, \text{wp}(B, Q))$
	<b>skip</b>	$A \sqcap B$	$\text{wp}(A, Q) \wedge \text{wp}(B, Q)$

Table 2: The guarded command language (left), along with the corresponding weakest precondition predicate transformer (right).

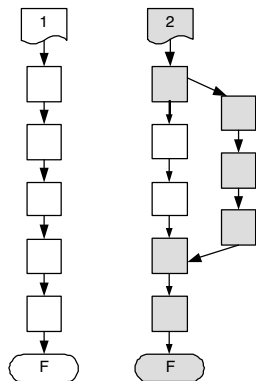


Figure 2: Different execution paths could end up in the same output states. The validation phase checks whether the new execution path explored by the candidate deviation input obtained in the deviation detection phase truly ends up in a different state.

### 3.2 Deviation Detection Phase

In this phase, we use a solver to find candidate inputs which may cause deviations. This phase takes as input the formulas  $f_1$  and  $f_2$  generated for the programs  $P_1$  and  $P_2$  in the formula extraction phase. We rewrite the variables in each formula so that they refer to the same input, but each to their own internal states.

We then query the solver whether the combined formula  $(f_1 \wedge \neg f_2) \vee (\neg f_1 \wedge f_2)$  is satisfiable, and if so, to provide an example that satisfies the combined formula. If the solver returns an example, then we have found an input that satisfies one program's formula, but not the other. If we had perfectly and fully modeled each program, and perfectly specified the post-condition to be that “the input results in a semantically equivalent output state”, then this input would be guaranteed to produce a semantically equivalent output state in one program, but not the other. Since we only consider one program path and do not perfectly specify the post-condition in this way, this input is only a *candidate deviation input*.

### 3.3 Validation Phase

Finally, we validate the generated candidate deviation inputs to determine whether they actually result in semantically different output states in the two implementations. As illustrated in Figure 2, it is possible that while an input does not satisfy the symbolic formula generated for a server, it actually does result in an identical or semantically equivalent output state.

We send each candidate deviation input to the implementations being examined, and compare their outputs to determine whether they result in semantically equivalent or semantically different output states.

In theory, this testing requires some domain knowledge about the protocol implemented by the binaries, to determine whether their outputs are semantically equivalent. In practice, we have found deviations that are quite obvious. Typically, the server whose symbolic formula is satisfied by the input produces a response similar to its response to the original input, and the server whose symbolic formula is not satisfied by the input produces an error message, drops the connection, etc.

## 4 Implementation

Our implementation consists of several components: a path recorder, the symbolic formula generator, the solver, and a validator. We describe each below.

**Collecting the trace.** The symbolic formula generator component is based on QEMU, a complete system emulator [10]. We use a modified version of QEMU, that has been enhanced with the ability to track how specified external inputs, such as keyboard or received network data are processed. The formula generator monitors the execution of a binary and records the execution trace, containing all instructions executed by the program and the information of their operands, such as their value and whether they are derived from specified external inputs. We start monitoring the execution before sending requests to the server and stop the trace when we observe a response from the server. We use a no-response timer

to stop the trace if no answer is observed from the server after a configurable amount of time.

**Symbolic formula generation.** We implemented our symbolic formula generator as part of our BitBlaze binary analysis platform [1]. The BitBlaze platform can parse executables and instruction traces, disassemble each instruction, and translate the instructions into the IR shown in Table 1. The entire platform consists of about 16,000 lines of C/C++ code and 28,000 lines of OCaml, with about 1,600 lines of OCaml specifically written for our approach.

**Solver.** We use STP [30, 31] as our solver. It is a decision procedure specialized in modeling bit-vectors. After taking our symbolic formula as input, it either outputs an input that can satisfy the formula, or decides that the formula is not satisfiable.

**Candidate deviation input validation.** Once a candidate deviation input has been returned by the solver, we need to validate it against both server implementations and monitor the output states. For this we have built small HTTP and NTP clients that read the inputs, send them over the network to the servers, and capture the responses, if any.

After sending candidate inputs to both implementations, we determine the output state by looking at the response sent from the server. For those protocols that contain some type of status code in the response, such as HTTP in the Status-Line, each different value of the status code represents a different output state for the server. For those protocols that do not contain a status code in the response, such as NTP, we define a generic *valid state* and consider the server to have reached that state, as a consequence of an input, if it sends any well-formed response to the input, independently of the values of the fields in the response.

In addition, we define three special output states: a *fatal state* that includes any behavior that is likely to cause the server to stop processing future queries such as a crash, reboot, halt or resource starvation, a *no-response state* that indicates that the server is not in the fatal state but still did not respond before a configurable timer expired, and a *malformed state* that includes any response from the server that is missing mandatory fields. This last state is needed because servers might send messages back to the client that do not follow the guidelines in the corresponding specification. For example several HTTP servers, such as Apache or Savant, might respond to an incorrect request with a raw message written into the socket, such as the string “IOError” without including

Server	Version	Type	Binary Size
Apache	2.2.4	HTTP server	4,344kB
Miniweb	0.8.1	HTTP server	528kB
Savant	3.1	HTTP server	280kB
NetTime	2.0 beta 7	NTP server	3,702kB
Ntpd	4.1.72	NTP server	192kB

Table 3: Different server implementations used in our evaluation.

the expected HTTP Status-Line such as “HTTP/1.1 400 Bad Request”.

## 5 Evaluation

We have evaluated our approach on two different protocols: HTTP and NTP. We selected these two protocols as representatives of two large families of protocols: text protocols (e.g. HTTP) and binary protocols (e.g. NTP). Text and binary protocols present significant differences in encoding, field ordering, and methods used to separate fields. Thus, it is valuable to study both families. In particular, we use three HTTP server implementations and two NTP server implementations, as shown in Table 3. All the implementations are Windows binaries and the evaluation is performed on a Linux host running Fedora Core 5.

The original inputs, which we need to send to the servers during the formula extraction phase to generate the execution traces, were obtained by capturing a network trace from one of our workstations and selecting all the HTTP and NTP requests that it contained. For each HTTP request in the trace, we send it to each of the HTTP servers and monitor its execution, generating an execution trace as output. We proceed similarly for each NTP request, obtaining an execution trace for each request/server pair. In Section 5.1, we show the deviations we discovered in the web servers, and in Section 5.2, the deviations we discovered in the NTP servers.

### 5.1 Deviations in Web Servers

This section shows the deviations we found among three web server implementations: Apache, Miniweb, and Savant. For brevity and clarity, we only show results for a specific HTTP query, which we find to be specially important because it discovered deviations between different server pairs. Figure 3 shows this query, which is an HTTP GET request for the file `/index.html`.

**Deviations detected.** For each server we first calculate a symbolic formula that represents how the server han-

**Original request:**

```

:                               /inde .html
:                               H   / . ..Ho t:
:                               . . . ....

```

Figure 3: One of the original HTTP requests we used to generate execution traces from all HTTP servers, during the formula extraction phase.

	$\neg f_A$	$\neg f_M$	$\neg f_S$
$f_A$	N/A	Case 1: unsatisfiable	Case 2: 5/0
$f_M$	Case 3: 5/5	N/A	Case 4: 5/5
$f_S$	Case 5: unsatisfiable	Case 6: unsatisfiable	N/A

Table 4: Summary of deviations found for the HTTP servers, including the number of candidate input queries requested to the solver and the number of deviations found. Each cell represents the results from one query to the solver and each query to the solver handles half of the combined formula for each server pair. For example Case 3 shows the results when querying the solver for  $(f_M \wedge \neg f_A)$  and the combined formula for the Apache-Miniweb pair is the disjunction of Cases 1 and 3.

dled the original HTTP request shown in Figure 3. We call these formulas:  $f_A$ ,  $f_S$ ,  $f_M$  for Apache, Savant and Miniweb respectively. Then, for each of the three possible server pairs: Apache-Miniweb, Apache-Savant and Savant-Miniweb, we calculate the combined formula as explained in Section 3. For example, for the Apache-Miniweb pair, the combined formula is  $(f_A \wedge \neg f_M) \vee (f_M \wedge \neg f_A)$ . To obtain more detailed information, we break the combined formula into two separates queries to the solver, one representing each side of the disjunction. For example, for the Apache-Miniweb pair, we query the solver twice: one for  $(f_A \wedge \neg f_M)$  and another time for  $(f_M \wedge \neg f_A)$ . The combined formula is the disjunction of the two responses from the solver.

Table 4 summarizes our results when sending the HTTP GET request in Figure 3 to the three servers. Each cell of the table represents a different query to the solver, that is, half of the combined formula for each server pair. Thus, the table has six possible cells. For example, the combined formula for the Apache-Miniweb pair, is shown as the disjunction of Cases 1 and 3.

Out of the six possible cases, the solver returned unsatisfiable for three of them (Cases 1, 5, and 6). For the remaining cases, where the solver was able to generate at least one candidate deviation input, we show two numbers in the format X/Y. The X value represents the number of different candidate deviation inputs we obtained from the solver, and the Y value represents the number of these candidate deviation inputs that actually generated semantically different output states when sent to the servers in the validation phase. Thus, the Y value represents the number of inputs that triggered a deviation.

In Case 2, none of the five candidate deviation inputs returned by the solver were able to generate semantically

different output states when sent to the servers, that is, no deviations were found. For Cases 3 and 4, all candidate deviation inputs triggered a deviation when sent to the servers during the validation phase. In both cases, the Miniweb server accepted some input that was rejected by the other server. We analyze these cases in more detail next.

#### Applications to error detection and fingerprint generation.

Figure 4 shows one of the deviations found for the Apache-Miniweb pair. It presents one of the candidate deviation inputs obtained from the solver in Case 3, and the responses received from both Apache and Miniweb when that candidate input was sent to them during the validation phase. The key difference is on the fifth byte of the candidate deviation input, whose original ASCII value represented a slash, indicating an absolute path. In the generated candidate deviation input, the byte has value 0xE8. We have confirmed that Miniweb does indeed accept any value on this byte. So, this deviation reflects an error by Miniweb: it ignores the first character of the requested URI and assumes it to be a slash, which is a deviation from the URI specification [16].

Figure 5 shows one of the deviations found for the Savant-Miniweb pair. It presents one of the candidate deviation inputs obtained from the solver in Case 4, including the responses received from both Savant and Miniweb when the candidate deviation input was sent to them during the validation phase. Again, the candidate deviation input has a different value on the fifth byte, but in this case the response from Savant is only a raw “File not found” string. Note that this string does not include the HTTP Status-Line, the first line in the response that includes the response code, as required by the HTTP spec-

**Candidate deviation input:**

```

:                                     .inde .html
:                                     .....HO. .
:                                     .....

```

**Miniweb response:**

```

H / .
  erver: iniweb
  ache control: no cache
  ...

```

**Apache response:**

```

H / .      ad e ue t
ate: at,    eb      : :
erver: pache/ . .    in
  ...

```

Figure 4: Example deviation found for Case 3, where Miniweb’s formula is satisfied while Apache’s isn’t. The figure includes the candidate deviation input being sent and the responses obtained from the servers, which show two different output states.

**Candidate deviation input:**

```

:                                     .inde .html
:                                     .....HO...
:                                     .....

```

**Miniweb response:**

```

H / .
  erver: iniweb
  ache control: no cache
  ...

```

**Savant response:**

```

file not found

```

Figure 5: Example deviation found for Case 4, where Miniweb’s formula is satisfied while Savant’s isn’t. The output states show that Miniweb accepts the input but Savant rejects it with a malformed response.

ification and can be considered malformed [27]. Thus, this deviation identifies an error though in this case both servers (i.e. Miniweb and Savant) are deviating from the HTTP specification.

Figure 6 shows another deviation found in Case 4 for the Savant-Miniweb pair. The HTTP specification mandates that the first line of an HTTP request must include a protocol version string. There are 3 possible valid values for this version string: “HTTP/1.1”, “HTTP/1.0”, and “HTTP/0.9”, corresponding to different versions of the HTTP protocol. However, we see that the candidate deviation input produced by the solver uses instead a different version string, “HTTP/\b.1”. Since Miniweb accepts this answer, it indicates that Miniweb is not properly verifying the values received on this field. On the other hand, Savant is sending an error to the client indicating an invalid HTTP version, which indicates that it is properly checking the value it received in the version field. This deviation shows another error in Miniweb’s implementation.

To summarize, in this section we have shown that our approach is able to discover multiple inputs that trigger deviations between real protocol implementations. We have presented detailed analysis of three of them, and

confirmed the deviations they trigger as errors. Out of the three inputs analyzed in detail, two of them can be attributed to be Miniweb’s implementation errors, while the other one was an implementation error by both Miniweb and Savant. The discovered inputs that trigger deviations can potentially be used as fingerprints to differentiate among these implementations.

## 5.2 Deviations in Time Servers

In this section we show our results for the NTP protocol using two different servers: NetTime [7] and Ntpd [13]. Again, for simplicity, we focus on a single request that we show in Figure 7. This request represents a simple query for time synchronization from a client. The request uses the Simple Network Time Protocol (SNTP) Version 4 protocol, which is a subset of NTP [38].

**Deviations detected.** First, we generate the symbolic formulas for both servers:  $f_T$  and  $f_N$  for NetTime and Ntpd respectively using the original request shown in Figure 7. Since we have one server pair, we need to query the solver twice. In Case 7, we query the solver for  $(f_N \wedge \neg f_T)$  and in Case 8 we query it for  $(f_T \wedge \neg f_N)$ .

**Candidate deviation input:**

```

:                                     /inde .html
:                                     H  /... ..Ho...
:                                     .....

```

**Miniweb response:**

```

H / .
erver: iniweb
ache control: no cache
...

```

**Savant response:**

```

H / .          nly . and . re ue t   upported
erver: avant/ .
ontent ype: te t/html
...

```

Figure 6: Another example deviation for Case 4, between Miniweb and Savant. The main different is on byte 21, which is part of the Version string. In this case Miniweb accepts the request but Savant rejects it.

**Original request:**

```

:  (e) fa
:
:                                     c e b a ca e a
:                                     LI   VN   MD

```

**Candidate deviation input:**

```

:  ( )
:
:                                     c e b a ca e a
:                                     LI   VN   MD

```

**NetTime response:**

```

:      f      fa
:  c e      c a c   a e c c   e b a ca e a
:  c e      e c   e

```

**Ntpd response:**

```

o re pon e

```

Figure 7: Example deviation obtained for the NTP servers. It includes the original request sent in the formula extraction phase, the candidate deviation input output by the solver, and the responses received from the servers, when replaying the candidate deviation input. Note that the output states are different since NetTime does send a response, while Ntpd does not.

The solver returns unsatisfiable for Case 7. For Case 8, the solver returns several candidate deviation inputs. Figure 7 presents one of the deviations found for Case 8. It presents the candidate deviation input returned by the solver, and the response obtained from both NTP servers when that candidate deviation input was sent to them during the validation phase.

**Applications to error detection and fingerprint generation.** The results in Figure 7 show that the candidate deviation input returned by the solver in Case 8 has different values at bytes 0, 2 and 3. First, bytes 2 and 3 have been zeroed out in the candidate deviation input. This is not relevant since these bytes represent the “Poll” and “Precision” fields and are only significant in messages sent by servers, not in the queries sent by the clients, and thus can be ignored.

The important difference is on byte 0, which is presented in detail on the right hand side of Figure 7. Byte

0 contains three fields: “Leap Indicator” (LI), “Version” (VN) and “Mode” (MD) fields. The difference with the original request is in the Version field. The candidate deviation input has a decimal value of 0 for this field (note that the field length is 3 bits), instead of the original decimal value of 4. When this candidate deviation input was sent to both servers, Ntpd ignored it, choosing not to respond, while NetTime responded with a version number with value 0. Thus, this candidate deviation input leads the two servers into semantically different output states.

We check the specification for this case to find out that a zero value for the Version field is reserved, and according to the latest specification should no longer be supported by current and future NTP/SNTP servers [38]. However, the previous specification states that the server should copy the version number received from the client in the request, into the response, without dictating any special handling for the zero value. Since both implementations seem to be following different versions of the

Program	Trace-to-IR time	% of Symbolic Instructions	IR-to-formula time	Formula Size
Apache	7.6s	3.9%	31.87s	49786
Miniweb	5.6s	1.0%	14.9s	25628
Savant	6.3s	2.2%	15.2s	24789
Ntpd	0.073s	0.1%	5.3s	1695
NetTime	0.75s	0.1%	4.3s	5059

Table 5: Execution time and formula size obtained during the formula extraction phase.

	Input Calculation Time
Apache - Miniweb	21.3s
Apache - Savant	11.8s
Savant - Miniweb	9.0s
NetTime - Ntpd	0.56s

Table 6: Execution time needed to calculate a candidate deviation input for each server pair.

specification, we cannot definitely assign this error to one of the specifications. Instead, this example shows that we can identify inconsistencies or ambiguity in protocol specifications. In addition, we can use this query as a fingerprint to differentiate between the two implementations.

### 5.3 Performance

In this section, we measure the execution time and the output size at different steps in our approach. The results from the formula extraction phase and the deviation detection phase are shown in Table 5 and Table 6, respectively. In Table 5, the column “Trace-to-IR time” shows the time spent in converting an execution trace into our IR program. The values show that the time spent to convert the execution trace is significantly larger for the web servers, when compared to the time spent on the NTP servers. This is likely due to a larger complexity of the HTTP protocol, specifically a larger number of conditions affecting the input. This is shown in the second column as the percentage of all instructions that operate on symbolic data, i.e., on data derived from the input. The “IR-to-formula time” column shows the time spent in generating a symbolic formula from the IR program. Finally, the “Formula Size” column shows the size of the generated symbolic formulas, measured by the number of nodes that they contain. The formula size shows again the larger complexity in the HTTP implementations, when compared to the NTP implementations.

In Table 6, we show the time used by the solver in the deviation detection phase to produce a candidate deviation input from the combined symbolic formula. The results show that our approach is very efficient in dis-

covering deviations. In many cases, we can discover deviation inputs between two implementations in approximately one minute. Fuzz testing approaches are likely to take much longer, since they usually need to test many more examples.

## 6 Discussion and Future Work

Our current implementation is only a first step. In this section we discuss some natural extensions that we plan to pursue in the future.

**Addressing other protocol interactions.** Currently, we have evaluated our approach over protocols that use request/response interactions (e.g. HTTP, NTP), where we examine the request being received by a server program. Note that our approach could be used in other scenarios as well. For example, with clients programs, we could analyze the response being received by the client. In protocol interactions involving multiple steps, we could consider the protocol output state to be the state of the program after the last step is finished.

**Covering rarely used paths.** Some errors are hidden in rarely used program paths and finding them can take multiple iterations in our approach. For each iteration, we need a protocol input that drives both implementations to semantically equivalent output states. These protocol inputs are usually obtained from a network trace. Thus, the more different inputs contained in the trace the more paths we can potentially cover. In addition, we can query the solver for multiple candidate deviation inputs, each time requiring the new candidate input to be different than the previous ones. The obtained candidate inputs often result in different paths. We have done work on symbolic execution techniques to explore multiple program paths and plan to apply those techniques here [2, 17].

**Creating formulas including multiple paths.** In this paper, we apply the weakest precondition on IR programs that contain a single program path, i.e., the processing of the original input by one implementation.

However, our weakest precondition algorithm is capable of handling IR programs containing multiple paths [19]. In the future, we plan to explore how to create formulas that include multiple paths.

**On-line formula generation.** Our current implementation for generating the symbolic formula works offline. We first record an execution trace for each implementation while it processes an input. Then, we process the execution trace by converting it into the IR representation, and computing the symbolic formula. Another alternative would be to generate the symbolic formulas in an on-line manner as the program performs operations on the received input, as in BitScope [2, 17].

## 7 Related Work

**Symbolic execution & weakest precondition.** Symbolic execution was first proposed by King [34], and has been used for a wide variety of problems including generating vulnerability signatures [18], automatic test case generation [32], proving the viability of evasion techniques [35], and finding bugs in programs [21, 47]. Weakest precondition was originally proposed for developing correct programs from the ground up [24, 26]. It has been used for different applications including finding bugs in programs [28] and for sound replay of application dialog [42].

**Static source code analysis.** Chen et al. [23] manually identify rules representing ordered sequences of security-relevant operations, and use model checking techniques to detect violations of those rules in software. Udrea et al. [45] use static source code analysis to check if a C implementation of a protocol matches a manually specified rule-based specification of its behavior.

Although these techniques are useful, our approach is quite different. Instead of comparing an implementation to a manually defined model, we compare implementations against each other. Another significant difference is that our approach works directly on binaries, and does not require access to the source code.

**Protocol error detection.** There has been considerable research on testing network protocol implementations, with heavy emphasis on automatically detecting errors in network protocols using fuzz testing [3–6, 9, 12, 14, 33, 37, 43, 46]. Fuzz testing is a technique in which random or semi-random inputs are generated and fed to the program under study, while monitoring for unexpected program output, usually an unexpected final state such as program crash or reboot.

Compared to fuzz testing, our approach is more efficient for discovering deviations since it requires testing far fewer inputs. It can detect deviations by comparing how two implementations process the same input, even if this input leads both implementation to semantically equivalent states. In contrast, fuzz testing techniques need observable differences between implementations to detect a deviation.

There is a line of research using model checking to find errors in protocol implementations. Musuvathi et al. [40, 41] use a model checker that operates directly on C and C++ code and use it to check for errors in TCP/IP and AODV implementations. Chaki et al. [22] build models from implementations and checks it against a specification model. Compared to our approach, these approaches need reference models to detect errors.

**Protocol fingerprinting.** There has also been previous research on protocol fingerprinting [25, 44] but available fingerprinting tools [8, 11, 15] use manually extracted fingerprints. More recently, automatic fingerprint generation techniques, working only on network input and output, have been proposed [20]. Our approach is different in that we use binary analysis to generate the candidate inputs.

## 8 Conclusion

In this paper, we have presented a novel approach to automatically detect deviations in the way different implementations of the same specification check and process their input. Our approach has several advantages: (1) by automatically building the symbolic formulas from the implementation, our approach is precisely truthful to the implementation; (2) automatically identifying the deviation by solving formulas generated from the two implementations enables us to find the needle in the haystack without having to try each straw (input) individually, thus a tremendous performance gain; (3) our approach works on binaries directly, i.e., without access to source code. We then show how to apply our automatic deviation techniques for automatic error detection and automatic fingerprint generation.

We have presented our prototype system to evaluate our techniques, and have used it to automatically discover deviations in multiple implementations of two different protocols: HTTP and NTP. Our results show that our approach successfully finds deviations between different implementations, including errors in input checking, and differences in the interpretation of the specification, which can be used as fingerprints.

## Acknowledgments

We would like to thank Heng Yin for his support on QEMU and Ivan Jager for his help in developing BitBlaze, our binary analysis platform. We would also like to thank Vijay Ganesh and David Dill for their support with STP, and the anonymous reviewers for their insightful comments.

This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0433540, No. 0448452, No. 0627511, and CCF-0424422. Partial support was also provided by the International Technology Alliance, and by the U.S. Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316, and under grant DAAD19-02-1-0389 through CyLab at Carnegie Mellon.

The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of ARO, NSF, or the U.S. Government or any of its agencies.

## References

- [1] The BitBlaze binary analysis platform. <http://www.cse.cmu.edu/~dbrumley/bitblaze>.
- [2] Bitscope. <http://www.cse.cmu.edu/chartwig/bitcope>.
- [3] IrcFuzz. <http://www.digitaldwarf.be/product/ircfuzz.c>.
- [4] ISIC: IP stack integrity checker. <http://www.packetfactory.net/project/>.
- [5] JBroFuzz. [http://www.owasp.org/index.php/category:\\_ro\\_u\\_](http://www.owasp.org/index.php/category:_ro_u_).
- [6] MangleMe. <http://lcamtuf.coredump.cx>.
- [7] NetTime. <http://nettime.sourceforge.net>.
- [8] Nmap. <http://www.insecure.org>.
- [9] Peach. <http://peachfuzzer.sourceforge.net>.
- [10] QEMU: an open source processor emulator. <http://www.qemu.org>.
- [11] Queso. <http://ftp.ceria.purdue.edu/pub/tool/unicanner/ueo>.
- [12] Spike. <http://www.immunity-sec.com/resource-free-software.html>.
- [13] Windows NTP server. <http://www.ee.udel.edu/mill/ntp/html/build/hint/winnt.html>.
- [14] Wireshark: fuzz testing tools. <http://wiki.wireshark.org/usage/testing>.
- [15] Xprobe. <http://www.yesecurity.com>.
- [16] BERNERS-LEE, T., FIELDING, R., AND MASINTER, L. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), 2005.
- [17] BRUMLEY, D., HARTWIG, C., LIANG, Z., NEWSOME, J., SONG, D., AND YIN, H. Towards automatically identifying trigger-based behavior in malware using symbolic execution and binary analysis. Tech. Rep. CMU-CS-07-105, Carnegie Mellon University School of Computer Science, 2007.
- [18] BRUMLEY, D., NEWSOME, J., SONG, D., W., H., AND JHA, S. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006).
- [19] BRUMLEY, D., WANG, H., JHA, S., AND SONG, D. Creating vulnerability signatures using weakest pre-conditions. In *Proceedings of the 2007 Symposium on Computer Security Foundations Symposium* (2007).
- [20] CABALLERO, J., VENKATARAMAN, S., POOSANKAM, P., KANG, M. G., SONG, D., AND BLUM, A. Fig: Automatic fingerprint generation. In *14th Annual Network and Distributed System Security Conference (NDSS)* (2007).
- [21] CADAR, C., GANESH, V., PAWLOWSKI, P., DILL, D., AND ENGLER, D. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)* (2006).
- [22] CHAKI, S., CLARKE, E., GROCE, A., JHA, S., AND VEITH, H. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)* (2003).
- [23] CHEN, H., AND WAGNER, D. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and Communications Security (CCS)* (2002).

- [24] COHEN, E. *Programming in the 1990's*. Springer-Verlag, 1990.
- [25] COMER, D., AND LIN, J. C. Probing TCP implementations. In *USENIX Summer 1994* (1994).
- [26] DIJKSTRA, E. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
- [27] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.
- [28] FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. Extended static checking for Java. In *ACM Conference on the Programming Language Design and Implementation (PLDI)* (2002).
- [29] FLANAGAN, C., AND SAXE, J. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th ACM Symposium on the Principles of Programming Languages (POPL)* (2001).
- [30] GANESH, V., AND DILL, D. STP: A decision procedure for bitvectors and arrays. <http://theory.stanford.edu/~vganesh/stp.html>.
- [31] GANESH, V., AND DILL, D. A decision procedure for bit-vectors and arrays. In *Proceedings of the Computer Aided Verification Conference* (2007).
- [32] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In *Proceedings of the 2005 Programming Language Design and Implementation Conference (PLDI)* (2005).
- [33] KAKSONEN, R. *A Functional Method for Assessing Protocol Implementation Security*. PhD thesis, Technical Research Centre of Finland, 2001.
- [34] KING, J. Symbolic execution and program testing. *Communications of the ACM* 19 (1976), 386–394.
- [35] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium* (2005).
- [36] LEINO, K. R. M. Efficient weakest preconditions. *Information Processing Letters* 93, 6 (2005), 281–288.
- [37] MARQUIS, S., DEAN, T. R., AND KNIGHT, S. SCL: a language for security testing of network applications. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research* (2005).
- [38] MILLS, D. Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI. RFC 4330 (Informational), 2006.
- [39] MUCHNICK, S. *Advanced Compiler Design and Implementation*. Academic Press, 1997.
- [40] MUSUVATHI, M., AND ENGLER, D. R. Model checking large network protocol implementations. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI)* (2004).
- [41] MUSUVATHI, M., PARK, D. Y., CHOU, A., ENGLER, D. R., , AND DILL, D. L. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (2002).
- [42] NEWSOME, J., BRUMLEY, D., FRANKLIN, J., AND SONG, D. Replayer: Automatic protocol replay by binary analysis. In *Proceedings of the 13<sup>th</sup> ACM Conference on Computer and Communications Security (CCS)* (2006).
- [43] OEHLERT, P. Violating assumptions with fuzzing. *IEEE Security and Privacy Magazine* 3, 2 (2005), 58 – 62.
- [44] PAXSON, V. Automated packet trace analysis of TCP implementations. In *ACM SIGCOMM 1997* (1997).
- [45] UDREA, O., LUMEZANU, C., AND FOSTER, J. S. Rule-based static analysis of network protocol implementations. In *Proceedings of the 15th USENIX Security Symposium* (2006).
- [46] XIAO, S., DENG, L., LI, S., AND WANG, X. Integrated tcp/ip protocol software testing for vulnerability detection. In *Proceedings of International Conference on Computer Networks and Mobile Computing* (2003).
- [47] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006).

# OSLO: Improving the security of Trusted Computing

Bernhard Kauer  
Technische Universität Dresden  
Department of Computer Science  
01062 Dresden, Germany  
kauer@os.inf.tu-dresden.de

## Abstract

*In this paper we describe bugs and ways to attack trusted computing systems based on a static root of trust such as Microsoft's Bitlocker. We propose to use the dynamic root of trust feature of newer x86 processors as this shortens the trust chain, can minimize the Trusted Computing Base of applications and is less vulnerable to TPM and BIOS attacks. To support our claim we implemented the Open Secure LOader (OSLO), the first publicly available bootloader based on AMDs `skinit` instruction.*

## 1 Introduction

An increasing number of Computing Platforms with a Trusted Platform Module (TPM) [33] are deployed. Applications using these chips are not widely used yet [5, 37]. This will change rapidly with the distribution of Microsoft's Bitlocker [2], a disk encryption utility which is part of Windows Vista Ultimate. As the trusted computing technology behind these applications is quite new, there is not much experience concerning the security of trusted computing systems. In this context we analyzed the security of TPMs, BIOSes and bootloaders that constitute the basic building blocks of trusted computing implementations. Furthermore, we propose a design that can improve the security of such implementations.

### 1.1 Trusted Computing

Trusted Computing [9, 23, 25, 33] is a technology that tries to answer two questions:

- Which software is running on a remote computer? (*Remote Attestation*)

- How to ensure that only a particular software stack can access a stored secret? (*Sealed Memory*)

Different scenarios can be built on top of trusted computing, for example, multi-factor authentication [37], hard disk encryption [2, 5] or the widely disputed Digital Rights Management. All of these applications are based on a small chip: the Trusted Platform Module (TPM).

### 1.2 Technical Background

As defined by the Trusted Computing Group (TCG), a TPM is a smartcard-like low performance cryptographic coprocessor. It is soldered<sup>1</sup> on various motherboards. In addition to cryptographic operations such as signing and hashing, a TPM can store hashes of the boot sequence in a set of Platform Configuration Registers (PCRs).

A PCR is a 160 bit wide register that can hold an SHA-1 hash. It cannot be directly written. Instead, it can only be modified using the `extend(x)` operation. This operation calculates the new value of a PCR as an SHA-1 hash of the concatenation of the old value and `x`. The `extend` operation is used to store a hash of a chain of loaded software in PCRs. The chain starts with the BIOS and includes Option ROMs<sup>2</sup>, Bootloader, OS and applications.

Using a challenge-response protocol, this *trust chain* can attest to a remote entity which software is running on the platform (*remote attestation*). Similarly it can be used to *seal* some data to a particular, not necessarily the currently running, software configuration. Unsealing the data is then only possible when this configuration was started. Figure 1 shows such a trust chain based on a Static Root of Trust for Measurement (SRTM), namely the BIOS.

$TPM \implies BIOS \implies OptionROMs \implies$   
 $BootLoader \implies OS \implies Application$

**Figure 1. Typical trust chain in a TC system**

### 1.3 Chain of Hashes

Three conditions must be met, to make a chain of hashes trustworthy:

1. The first code running and extending PCRs after a platform reset (called SRTM) is trustworthy and cannot be replaced.
2. The PCRs are not resettable, without passing control to trusted code.
3. The chain is contiguous. There is no code in-between that is executed but not hashed.

The reasons behind these conditions are the following: If the initial code is not trustworthy or can be replaced by untrustworthy code, it cannot be guaranteed that any hash value is correct. This code can in fact modify any later running software to prevent the undesirable hashing. The second condition is quite similar and can be seen as a generalization of the first one. If PCRs are reset and untrustworthy code is running then any chain of hashes can be fabricated. The first two points describe the beginning of the trust chain. The third point is needed to form a contiguous chain by recursion. It forces the condition that every program occupying the machine must be hashed, before it is executed. Otherwise, the trust chain is interrupted and unmeasured code can be running. Every program using sealed memory has to trust the code running before it to not open a hole in the chain. Similarly, a remote entity needs to find out during an attestation whether the trust chain presented by a trusted computing platform contains any hole in which untrusted code could be run. We will see later how current implementations do not meet the three conditions.

### Organization

This paper is structured as follows. We describe bugs and ways to attack trusted computing systems based on SRTM in the next section. After that we present the design and describe the implementation of OSLO. A section evaluating the security achievements follows. The last section proposes future work and concludes.

## 2 Security Analysis

### 2.1 Bootloader Bugs

We look at the three publicly available TPM-enabled bootloaders and analyze whether they violate the third condition of a trust chain, executing code that is not hashed.

The very first publicly available trusted bootloader was part of the Bear project from Dartmouth College [19, 20]. They enhanced Linux with a security module called Enforcer. This module checks for modification of files and uses the TPM to seal a secret key of an encrypted filesystem. To boot the system they used a modified version of LILO [7]. They extend LILO in two ways: the Master Boot Record hashes the rest of LILO and the loaded Linux kernel image is also hashed. Only the last part of the image, containing the kernel itself, is hashed here. But the first part of the image, containing the real-mode setup code, is executed. Hence, this violates the third condition. A fix for this bug would be to hash every sector which gets loaded.

A second trusted bootloader is a patched GRUB v0.97 from IBM Japan [21, 36]. This bootloader is used in IBMs Integrity Measurement Architecture [28]. It has the same security flaw as our own experiments with a TCG enabled GRUB [16]: it loads files twice, first for extraction and later for hashing into a PCR. A cause for this bug lies certainly in the structure of GRUB. GRUB loads and extracts a kernel image at the same time instead of loading them completely into memory and extracting them afterwards. This leads to the situation that measuring the file independently from loading is the easiest way for a programmer to add TCG support to GRUB. Such an implementation is unfortunately incorrect. As program code is loaded twice from disk or from a remote host over the network, an attacker who has physical access either to the disk or to the network can send different data at the second time. This violates again the third condition, as hashed and executed code may differ.

Another GRUB based trusted bootloader called TrustedGRUB [35] solves this issue in a recent version by moving the hash code to a lower level. Hashing is simply done on each `read()` call that loads data from disk or network, before the actual data is returned to the caller. The hash is then used after loading a kernel to extend a PCR.

The current version 1.0-rc5 of TrustedGRUB (August 2006) contains at least two other bugs. The hashing of its own code when starting from hard disk is broken. The corresponding PCR is never extended and always zero. Furthermore TrustedGRUB never contained any code to

use it securely from a CD. Nevertheless, it is used on a couple of LiveCDs [6].

All publicly available TPM-enabled bootloaders violate the third assumption, which makes systems booted by them unable to prove their trustworthiness. To analyze this it was not necessary to look at more sophisticated attack points such as missing range checks or buffer overflows. Both of these will become more interesting if the aforementioned bugs are fixed.

## 2.2 TPM Reset

In July 2004 we discovered that setting the reset bit in a control register of a v1.1 TPM<sup>3</sup> resets the chip without resetting the whole platform. This violates the second condition. As it results in default PCR values, this breaks the *remote attestation* and *sealing* features of those chips: Any PCR value can be reproduced without the opportunity for a remote entity to see the difference via remote attestation. Unsealing protected secrets of a security critical program is possible after resetting as well. The reset feature was added for maintenance reasons but does not have broad security consequences, because sealing and remote attestation are not used in any product application with v1.1 chips. Instead the chips are solely used as smartcard for signing and key management.

This case demonstrates the security risk of a resettable TPM. As other chips have different interfaces and can therefore not be reset in the same way, we experimented with a simple hardware attack. The Low Pin Count (LPC) bus was the point of attack. Most TPMs are connected to the southbridge through it and the bus has a separate reset line. We used different TPMs on external daughterboards for this experiment.

By physically connecting the LRESET# pin to ground we were able to perform a reset of the chip itself. We separated the pin from the bus as otherwise the PS/2 keyboard controller received such a reset signal, too. We had to reinitialize the chip which we did by reloading the driver and then sending a `TPMStartup(TPM_CLEAR)` to the chip. This process gave us an activated and enabled TPM in a state normally only visible to the BIOS: As expected all PCRs were in their default state. We presume that this attack could be mounted against any TPM in a similar way.

The simplicity of the reset makes this hardware attack a threat to trusted computing systems. In particular in use cases where physical access, for example, through theft, can not be excluded. This attack also affects another use case of trusted computing, the widely disputed Digital Rights Management scenario where the owner of a device is untrusted and can use the system unin-

tendedly.

We have to admit that the TCG does not claim to protect against hardware attacks. But scenarios using trusted computing technology have to be aware of these restrictions.

## 2.3 BIOS Attack

We have shown that bootloader and TPM implementations have some weaknesses. Now we look at the entity in-between them: the BIOS.

The BIOS contains the Core Root of Trust for Measurement (CRTM), a piece of code that extends PCR 0 initially. A CRTM has only to be exchanged with vendor signed code. Currently, the CRTM of many machines is freely patchable. It is stored in flash and no signature checking is performed on updates. This violates the first condition needed by a trust chain.

We used a HP nx6325, a recent business notebook with a TPM v1.2, for this experiment. The fact that the BIOS is flashed from a raw image eased an attack. Other vendors are checking a hash before flashing the image to avoid transmission errors, a feature that is missing here. Checking a hash is irrelevant from a security point of view but it would make the following steps slightly more complicated, as we would have to recalculate the correct hash value.

The part of the BIOS we choose to patch is the TPM driver. This has the advantage that all commands to the TPM, whether they come from the CRTM or from a bootloader through the `INT 1Ah` interface, can be intercepted. Our BIOS has only a memory-present TPM driver. These drivers need access to main memory for execution and can therefore only run after the BIOS has initialized the RAM. The interface of the TPM drivers are defined in the TCG PC client specification for conventional BIOS [34]. The function that we want to disable is `MPTPMTransmit()` which transmits commands to the TPM. We found the TPM driver in the BIOS binary quite easily. Strings like 'TPM' and the magic number of the code block as well as characteristic mnemonics (e.g., `in` and `out`) in the disassembly point to it.

Figure 2 shows the start of the BIOS TPM driver. It starts with a magic number and entry point, both as defined in the specification. The code itself starts at address `0x28`. We now search for an instruction that allows us to disable `MPTPMTransmit()`. The first instructions of the driver are quite uninteresting. They just save some registers to the stack and calculate the drivers starting address in register `edi` in order to make the code position independent. The first interesting instruction is the comparison at address `0x3a`. By look-

```

0:  aa 55          /* magic number */
4:  28 00          /* entry point */
...
28: 57            push    %edi
29: 56            push    %esi
2a: 53            push    %ebx
2b: 33 ff          xor     %edi,%edi
2d: e8 00 00       call    0x32
    00 00
32: 5f            pop     %edi
33: 81 ef 33       sub     $0x33,%edi
    00 00
39: 47            inc     %edi
3a: 3c 04          cmp     $0x4,%al
3c: 74 23          je      0x61
3e: 3c 01          cmp     $0x1,%al
40: 74 0a          je      0x4c
...

```

**Figure 2. Start of BIOS TPM driver**

ing further into the disassembly we found out that this instruction is part of the branch where the code distinguishes between `MPTPMTransmit()` (where `al=4`) and other functions. By changing this comparison to `cmp $0x14,%al`, which just requires to flip a single bit, we can avoid that the branch at `0x3c` is taken and any command is transmitted to the TPM. An error code is returned to the caller instead.

We now have to flash the BIOS with this modified image. As there is no hash of the BIOS image checked during flashing we use the normal BIOS update procedure. After a reboot we have a TPM in its default power-on state, without any PCR extensions.

The ability to easily exchange the CRTM violates the TCG specifications. A result of this bug is that the trust into these machines can not be brought back anymore without an expensive certification process.

## 2.4 Summary

We found weaknesses in bootloaders and the possibility of a simple hardware attack against TPMs. Furthermore by just flipping a single bit we disabled the CRTM and any PCR extension from the BIOS. These cases show that current implementations do not meet all three conditions of a trust chain.

In summary, we conclude that current BIOSes and bootloaders are not able to start systems in a trustworthy manner. Moreover, TPMs are not protected against resets.

*TPM ⇒ OSLO ⇒ OS ⇒ Application*

**Figure 3. Trust chain with a Dynamic Root of Trust for Measurement (DRTM)**

## 3 Design and Implementation of OSLO

### 3.1 Using a DRTM

The main idea behind a secure system with a resettable TPM, an untrusted BIOS and a buggy bootloader, is to use a Dynamic Root of Trust for Measurement (DRTM). A DRTM effectively removes the BIOS, OptionROMs and Bootloaders from the trust chain (cf. Figure 3).

With a DRTM, the CPU can reset the PCR 17 at any time. This is provided through a new instruction that atomically initializes the CPU, loads a piece of code called Secure Loader (SL) into its cache, sends the code to the TPM to extend the reseted PCR 17, and transfers control to the SL.

A design based on a DRTM is not vulnerable to the TPM reset attack because of a TPM property that can be easily missed. A TPM can distinguish between a reset and a DRTM due to CPU and chipset support. A reset of the TPM sets all PCRs to default values, which is “0” for the PCRs 0 - 16 and “-1” for PCR 17. Only a DRTM, with its special bus cycles, will reset the PCR 17 to “0” and immediately extend it with the hash of the SL. Therefore, an attacker is unable to reset PCR 17 to “0” and fake other platform configurations. Only by executing the `skinit` instruction it is possible to put the hash of an SL into PCR 17. An attacker can not hash an SL and directly afterwards executing code outside of it, since `skinit` jumps directly to the SL.

An SL is also not affected by the BIOS attack. With the presence of a DRTM, the BIOS need not be trusted anymore to protect its CRTM and hash itself into the TPM. Nevertheless, a statement that claims the BIOS can be fully untrusted is oversimplified: We still have to trust the BIOS for providing the System Management Mode (SMM) code as well as correct ACPI tables. As both can be security critical, a hash of them should be incorporated at boot time into a PCR by the operating system.

### 3.2 Implementation

AMD provides a DRTM with its `skinit` instruction which was introduced with the AMD-V extension [1]. On Intel CPUs, the Trusted Execution Technol-

ogy (TET) includes a DRTM with the `sender` instruction [9, 14]. AMD was generous to provide us with an AMD-V platform nearly one year earlier than we were able to buy an Intel TET platform.

Our implementation, called OSLO (Open Secure LOader), is written in C with some small parts in assembler. As OSLO is part of the Trusted Computing Base (TCB) of all applications, we wanted to minimize the binary and source code size. Furthermore, we had to avoid any BIOS call, as otherwise the BIOS would be part of the TCB again.

OSLO is started as kernel from a multi-boot compliant [22] loader. It initializes the TPM to be able to extend a PCR with the hashes of further modules. After that other processors are stopped. This is required before executing `skinit` and inhibits potential interferences during the secure startup procedure. For example, malicious code running on a second CPU could modify the instructions of the Secure Loader. The cache consistency protocol would then propagate the changes to the other processor.

Since the needed platform initialization is done, OSLO can now switch to the “secure mode” by executing `skinit`. Before starting the first module as a new kernel, OSLO hashes every module that is preloaded from the parent boot-loader.

We used chainloading via the multiboot specification to be flexible with respect to the operating system OSLO loads and who can load OSLO. Normally, this will be a multiboot-compliant loader started by the BIOS such as GRUB or SysLinux [31] but loading OSLO from the Linux `kexec` environment [17] should also be possible.

As we could not rely on the BIOS for talking to the TPM, we also implemented our own TPM driver for v1.2 TPMs. As all of these TPMs should follow the TPM interface specification (TIS) only a single driver was needed. Using this memory mapped interface is, compared to the different interfaces needed to talk to the v1.1 TPMs, rather simple. Therefore our TPM driver consists of only 70 lines of code.

Currently two features of OSLO are still unimplemented:

- protection against direct memory access (DMA) from malicious devices, and
- extension of the TPM event log for remote attestation.

The TPM event log is used to ease remote attestation. It can store hashes used as input for `extend` and optionally a string describing them. The log provides a breakdown of the PCR value into smaller known pieces. It is itself not security critical and therefore not protected

by the bootloader or the operating system. An attacker can only perform Denial of Service attacks by for example overwriting the log. It is not possible to compromise the security of a remote attestation by modifying the log. The TPM event log makes it much easier for a remote entity to check a reported hash values against a list of good known values, for example if the order of the `extends` is not fixed. OSLO should extend the event log to support applications relying on it for remote attestation.

The source code of OSLO is available under the terms of the GPL [24]. The source includes three additional tools that can be multi-boot loaded after OSLO: **Beirut** to hash command lines, **Pamplona** to revert the steps done by `skinit` for booting OSLO unaware OSes, and **Munich** to start Linux from a multiboot environment.

### 3.3 Lessons Learned

We have learned two lessons while implementing OSLO:

- It is hard to write secure initialization code, and
- a secure loader needs to have platform specific knowledge.

An example of the first lesson is our experience with the initialization of the Device Exclusion Vector (DEV) on AMD CPUs. A DEV is a bitvector in physical memory that consists of one bit per physical 4k-page. A bit in this vector decides whether device based DMA transfers to or from the corresponding page is allowed. DEVs could be cached in the chipset for performance reasons. We found out that the DEV initialization, if it is done in the naive way, contains a race condition.

DEV initialization is normally done in two steps: Enable the appropriate bits in the vector to protect itself and then flushing the chipset internal DEV cache. As these two operations are not atomic, a malicious device could change the DEV using DMA just before the vector is loaded into the DEV cache. An implementation has to find a workaround for this race. A secure way to initialize DEV protection is, for example, to use an intermediate DEV in the 64k of the secure loader thereby protecting the initialization of a final DEV.

The second point is a little bit more complicated. DEVs can only protect against DMA from a device. If someone puts an operating system he wants to start with OSLO into device memory it cannot be protected from a malicious device. The OS is loaded and hashed by OSLO as if it would reside in RAM, but if it is read the

Name	size	OSLO sha1	sha1sum
kernel	1.2 MB	0.070 sec	0.020 sec
initrd	4.2 MB	0.245 sec	0.064 sec
sum	5.4 MB	0.315 sec	0.084 sec

**Figure 4. Performance of hashing a Linux kernel and Initrd**

Name	LOC	binary in kb	gzip in kb
BIOS HP	-	1024	491
GRUB v0.97	19600	98	55
OSLO v0.4.2	1534	4.1	2.9

**Figure 5. Size of BIOS, GRUB and OSLO**

second time, e.g., on ELF decoding or execution, it is requested from the device memory again. Because we do not trust a device to leave its memory unmodified, we cannot be sure that the code that is executed is identical to the hashed one. As a consequence we can only protect, hash and start modules that are located in RAM. A secure loader therefore needs a reliable method to detect the distinction between RAM and device memory.

## 4 Evaluation

One of our design goals for OSLO was a minimal TCB size. Reducing the TCB is suitable for security sensitive applications as it increases the understandability and minimizes the number of possible bugs [30]. Furthermore, the process of formal verification will benefit from it. We achieved a minimal TCB by using two techniques: reducing functionality and trading size with performance penalties.

An example for the first is that we do not rely on external libc code but use functions with limited functionality like `out_string()` instead of a full featured `printf()` implementation.

We also implemented our own SHA-1 code trading size for performance. This resulted in an SHA-1 implementation that compiles with gcc-3.4 to less than 512 bytes. This is only a quarter of the size compared with a performance optimized version such as the one from the Linux kernel. This, on the other hand, makes the hash much slower. The Linux version has a throughput which is three to four times higher, due to, e.g., loop unrolling.

Figure 4 shows that booting linux with our SHA-1 implementation takes 0.315 seconds compared to 0.084 seconds for a heavily optimized sha1sum version. As booting a system usually takes minutes a performance penalty of 0.231 seconds is acceptable here.

Figure 5 shows the source and binary sizes for BIOS,

GRUB and OSLO. We also give the size of gzip compressed binaries in this table as this reduces the effect of empty sections in the images. Unfortunately, the source code of the HP BIOS is not available. A similar but older Award BIOS consists of around 150 thousand lines of assembler code. The numbers given for GRUB do not include the drivers used to boot from a network. Adding them would nearly double the given numbers.

OSLO is an order of magnitude smaller than GRUB and two orders of magnitude smaller than the BIOS we examined. If we presume the principle *more code equals more bugs* and neglect the effect of a code size optimizing compiler, we can deduce that OSLO has a significantly smaller number of bugs due to its size compared to GRUB or the BIOS.

One could argue that in an ordinary system like Windows or Linux, where the TCB of an application consists of million lines of code with programs consuming tens or hundreds of megabytes, the size of GRUB and the BIOS does not matter. That is perhaps true, but as the trend in secure systems goes to small kernels and hypervisors [10, 13, 29, 32], architectures like L4/NIZZA or Xen can very well benefit from the TCB reduction through OSLO.

In summary, OSLO promises a smaller attack surface due to its minimal size and since it uses a DRTM mitigates the **TPM reset** and the **BIOS attacks** as outlined in Section 3.1.

## 5 Related Work

Previous research showed the vulnerability of trusted computing platforms against hardware attacks. Kursawe et al. [18] eavesdrop on the LPC bus to capture and analyse the communication between the CPU and the TPM. They only perform a passive attack, but describe that an active hardware attack on the LPC bus could be used to fool the TPM about the platform state. Untrusted code can then pretend to the TPM to be a DRTM.

Limitations of the trusted computing specification and its implementations are described in the literature multiple times. Bruschi et al. [4] showed that an authorization protocol of TPMs is vulnerable to replay attacks. Sadeghi et al. [27] reported that many TPM implementations do not meet the TCG specification. Garriss et al. [8] found out that a public computing kiosk that uses remote attestation to prove which software is running is vulnerable to boot-between attestation attacks. They suggest a reboot counter in the TPM to make reboots visible to remote parties. Such a counter will not help against our TPM reset attack as it needs to detect whether a TPM was switched on later than the whole platform<sup>4</sup>, a property a reboot counter cannot achieve.

There are more sophisticated BIOS attacks mentioned in the literature. Heasman [12], for example, showed at the Blackhat Federal 2006 that a rootkit can be hidden in ACPI code which is usually stored in the BIOS. In a subsequent paper [11], he describes how a rootkit can persist in a system with a secured BIOS by using other flash chips. In both cases only TPM-less systems were considered. By combining our attack to disable the CRTM with Heasman's work it seems possible to hide a rootkit in the BIOS but report correct hash values to the TPM.

To generally prevent BIOS attacks, Phoenix Technologies offers a firmware called TrustedCore [26] that allows only signed updates. Intel Active Management Technology [15] has also this feature.

Sailer et al. [28] describe an architecture for an integrity measurement system for Linux using a static root of trust. As they focus on the enhancements of the operating system, the architecture is not limited to an SRTM. There implementation could easily benefit from the smaller attack surface of a secure loader like OSLO.

## 6 Future Work and Conclusion

OSLO is not feature complete yet. We plan to finish the implementation of the DMA protection. Moreover, we want to add ACPI event-log support. This should allow the integration of OSLO into larger projects that use the event-log for remote attestation.

A port of OSLO to use the `sender` instruction on an Intel TET platform could demonstrate that the multiboot chainloader design is portable or show that `sender` implies an integrated design as it is proposed for Xen [38].

The search for new attack points of other trusted computing implementations is also part of our future work.

It was not necessary to look at more sophisticated attack points such as buffer overflows or the strength of cryptographic algorithms to find the bugs and attacks we presented in this paper. If we compare this to a similar analysis of another secure system, such as the one of an RFID chip [3], we have to conclude that current trusted computing implementations are not resilient to even simple attacks. Moreover, the current implementations do not meet the assumptions of a secure design. Even a small bug in them can compromise the additional security obtained by a TPM.

We suspect that most of the platforms are vulnerable to the TPM reset and many of them to the BIOS attack. As a consequence the software still based on an SRTM, such as Microsoft's Bitlocker, cannot provide secure TPM-driven encryption and attestation on these systems.

A switch to a DRTM based OSLO-like approach can shorten the trust chain, minimize the TCB, and is less vulnerable to TPM and BIOS attacks.

## Acknowledgements

We would like to thank Hermann Härtig, Michael Peter, Udo Steinberg, Neal Walfield, Carsten Weinhold, and Björn Döbel for their comments. Additionally we would like to thank Adrian Perrig, Jonathan McCune and the reviewers for their suggestions to improve the paper. Special thanks go to Adam Lackorzynski for providing the hardware in time.

## Notes

<sup>1</sup>There exist also TPMs on daughterboards. Their security value is limited as exchanging them is quite easy.

<sup>2</sup>Firmware on adapter cards

<sup>3</sup>It would be quite unfair to disclose the vendor name here.

<sup>4</sup>e.g., by holding the reset line of a TPM while powering the machine up

## References

- [1] AMD. Secure Virtual Machine Architecture Reference Manual, May 2005.
- [2] BitLocker Drive Encryption: Technical Overview. URL: <http://technet.microsoft.com/en-us/windowsvista/aa906017.aspx>.
- [3] Steve Bono, Matthew Green, Adam Stubblefield, Ari Juels, Avi Rubin, and Michael Szydlo. Security analysis of a cryptographically-enabled RFID device. In *USENIX Security Symposium*, Baltimore, Maryland, USA, July 2005. USENIX.
- [4] D. Bruschi, L. Cavallaro, A. Lanzi, and M. Monga. Attacking a Trusted Computing Platform - Improving the Security of the TCG Specification. Technical Report RT 05-05, Università degli Studi di Milano, Milano MI, Italy, May 2005.
- [5] eCryptfs: An Enterprise-class Cryptographic Filesystem for Linux. URL: <http://ecryptfs.sourceforge.net>.
- [6] EMSCB downloads. URL: <http://www.emscb.com/content/pages/turaya.downloads>.
- [7] Enforcer Project. URL: <http://enforcer.sourceforge.net>.

- [8] Scott Garriss, Ramón Cáceres, Stefan Berger, Reiner Sailer, Leendert van Doorn, and Xiaolan Zhang. Towards Trustworthy Kiosk Computing. In *"Proceedings of the 8th IEEE Workshop on Mobile Computing Systems & Applications (HotMobile 2007)"*. IEEE Computer Society Press, February 2007.
- [9] David Grawrock. *The Intel Safer Computing Initiative*. Intel Press, January 2006.
- [10] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza secure-system architecture. In *Proceedings of the 1st International Conference on Collaborative Computing: Networking, Applications and Work-sharing (CollaborateCom 2005)*, December 2005.
- [11] John Heasman. Implementing and Detecting a PCI Rootkit. November 2006.
- [12] John Heasman. Implementing and Detecting an ACPI Rootkit. In *BlackHat Federal*, January 2006.
- [13] Christian Helmuth, Alexander Warg, and Norman Feske. Mikro-SINA—Hands-on Experiences with the Nizza Security Architecture. In *Proceedings of the D.A.CH Security 2005*, Darmstadt, Germany, March 2005.
- [14] Intel Corporation. LaGrande technology preliminary architecture specification. Intel Publication no. D52212, May 2006.
- [15] Intel Advanced Management Technology. URL: <http://www.intel.com/technology/manage/iamt>.
- [16] Bernhard Kauer. Authenticated Booting for L4. Study thesis, TU Dresden, November 2004.
- [17] Kexec Article. URL: <http://lwn.net/Articles/15468>.
- [18] Klaus Kursawe, Dries Schellekens, and Bart Preneel. Analyzing trusted platform communication. In *ECRYPT Workshop, CRASH - Cryptographic Advances in Secure Hardware*, September 2005.
- [19] Rich MacDonald, Sean W. Smith, John Marchesini, and Omen Wild. Bear: An Open-Source Virtual Secure Coprocessor based on TCPA. Technical Report TR2003-471, Dartmouth College, Hanover, NH, August 2003.
- [20] John Marchesini, Sean W. Smith, Omen Wild, and Rich MacDonald. Experimenting with tcpa/tcg hardware, or: How i learned to stop worrying and love the bear. Technical Report TR2003-476, Dartmouth College, Hanover, NH, December 2003.
- [21] H. Maruyama, F. Seliger, N. Nagaratnam, T. Ebringer, S. Munetoh, S. Yoshihama, and T. Nakamura. Trusted Platform on Demand. Technical Report RT0564, IBM Corporation, February 2004.
- [22] Multiboot Specification. URL: <http://www.gnu.org/software/grub/manual/multiboot/multiboot.txt>.
- [23] Chris J. Mitchell, editor. *Trusted Computing*. IEE, London, Nov 2005.
- [24] OSLO - Open Secure LOader. URL: <http://os.inf.tu-dresden.de/~kauer/oslo>.
- [25] Siani Pearson, editor. *Trusted Computing Platforms*. Prentice Hall International, Aug 2002.
- [26] Phoenix Technologies, TrustedCore. URL: <http://www.phoenix.com/en/Products/Core+System+Software/TrustedCore>.
- [27] Ahmad-Reza Sadeghi, Marcel Selhorst, Christian Stübke, Christian Wachsmann, and Marcel Winandy. TCG Inside? - A Note on TPM Specification Compliance. In *The First ACM Workshop on Scalable Trusted Computing (STC'06)*, November 2006.
- [28] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the USENIX Security Symposium*, August 2004.
- [29] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramón Cáceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *ACSAC*, pages 276–285, 2005.
- [30] Lenin Singaravelu, Calton Pu, Hermann Hartig, and Christian Helmuth. Reducing tcb complexity for security-sensitive applications: Three case studies. In *EuroSys 2006*, April 2006.
- [31] SYSLINUX Project. URL: <http://syslinux.zytor.com>.

- [32] Richard Ta-Min, Lionel Litty, and David Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, November 2006.
- [33] TCG: Trusted Computing Group. URL: <https://www.trustedcomputinggroup.org>.
- [34] TCG PC Client Implementation Specification for Conventional BIOS. URL: <https://www.trustedcomputinggroup.org/specs/PCClient>.
- [35] TrustedGRUB. URL: [http://www.prosec.rub.de/trusted\\_grub.html](http://www.prosec.rub.de/trusted_grub.html).
- [36] GRUB TCG Patch to support Trusted Boot. URL: <http://trousers.sourceforge.net/grub.html>.
- [37] Wave's Embassy Security Center. URL: <http://www.wave.com/products/esc.html>.
- [38] [Xen-devel] Intel(R) LaGrande Technology support. URL: <http://lists.xensource.com/archives/html/xense-devel/2006-09/msg00047.html>.



# Secretly Monopolizing the CPU Without Superuser Privileges

Dan Tsafir<sup>◇†</sup>   Yoav Etsion<sup>◇</sup>   Dror G. Feitelson<sup>◇</sup>

<sup>◇</sup>*School of Computer Science and Engineering  
The Hebrew University  
91904 Jerusalem, Israel  
{dants, etsman, feit}@cs.huji.ac.il*

<sup>†</sup>*IBM T. J. Watson Research Center  
P. O. Box 218  
Yorktown Heights, NY 10598  
dants@us.ibm.com*

## Abstract

We describe a “cheat” attack, allowing an ordinary process to hijack any desirable percentage of the CPU cycles without requiring superuser/administrator privileges. Moreover, the nature of the attack is such that, at least in some systems, listing the active processes will erroneously show the cheating process as not using any CPU resources: the “missing” cycles would either be attributed to some other process or not be reported at all (if the machine is otherwise idle). Thus, certain malicious operations generally believed to have required overcoming the hardships of obtaining root access and installing a rootkit, can actually be launched by non-privileged users in a straightforward manner, thereby making the job of a malicious adversary that much easier. We show that most major general-purpose operating systems are vulnerable to the cheat attack, due to a combination of how they account for CPU usage and how they use this information to prioritize competing processes. Furthermore, recent scheduler changes attempting to better support interactive workloads increase the vulnerability to the attack, and naive steps taken by certain systems to reduce the danger are easily circumvented. We show that the attack can nevertheless be defeated, and we demonstrate this by implementing a patch for Linux that eliminates the problem with negligible overhead.

## Prologue

Some of the ideas underlying the cheat attack were implemented by Tsutomu Shimomura circa 1980 at Princeton, but it seems there is no published or detailed essay on the topic, nor *any* mention of it on the web [54]. Related publications deal solely with the fact that general-purpose CPU accounting can be inaccurate, but never conceive this can be somehow maliciously exploited (see Section 2.3). Recent trends in mainstream schedulers render a discussion of the attack especially relevant.

## 1 Introduction

An *attacker* can be defined as one that aspires to perform actions “resulting [in the] violation of the explicit or implicit security policy of a system”, which if successful, constitute a *breach* [31]. Under this definition, the said actions may be divided into two classes. One is of *hostile actions*, e.g. unlawful reading of sensitive data, spamming, launching of DDoS attacks, etc. The other is of *concealment actions*. These are meant to prevent the hostile actions from being discovered, in an effort to prolong the duration in which the compromised machine can be used for hostile purposes. While not hostile, concealment actions fall under the above definitions of “attack” and “breach”, as they are in violation of any reasonable security policy.

The “cheat” attack we describe embodies both a hostile and a concealment aspect. In a nutshell, the attack allows to implement a cheat utility such that invoking

```
cheat p prog
```

would run the program `prog` in such a way that it is allocated exactly  $p$  percent of the CPU cycles. The hostile aspect is that  $p$  can be arbitrarily big (e.g. 95%), but `prog` would still get that many cycles, regardless of the presence of competing applications and the fairness policy of the system. The concealment aspect is that `prog` would erroneously appear as consuming 0% CPU in monitoring tools like `ps`, `top`, `xosview`, etc. In other words, the cheat attack allows a program to (1) consume CPU cycles in a secretive manner, and (2) consume as many of these as it wants. This is similar to the common security breach scenario where an attacker manages to obtain superuser privileges on the compromised machine, and uses these privileges to engage in hostile activities and to conceal them. But in contrast to this common scenario, the cheat attack requires no special privileges. Rather, it can be launched by regular users, deeming this important line of defense (of obtaining root or superuser privileges) as irrelevant, and making the job of the attacker significantly easier.

Concealment actions are typically associated with *rootkits*, consisting of “a set of programs and code that allows a permanent or consistent, undetectable presence on a computer” [25]. After breaking into a computer and obtaining root access, the intruder installs a rootkit to maintain such access, hide traces of it, and exploit it. Thus, ordinarily, the ability to perform concealment actions (the rootkit) is the result of a hostile action (the break-in). In contrast, with the cheat attack, it is exactly the opposite: the concealment action (the ability to appear as consuming 0% CPU) is actually what makes it possible to perform the hostile action (of monopolizing the CPU regardless of the system’s fairness policy). We therefore begin by introducing the OS mechanism that allows a non-privileged application to conceal the fact it is using the CPU.

## 1.1 Operating System Ticks

A general-purpose operating system (GPOS) typically maintains control by using periodic clock interrupts. This practice started at the 1960s [12] and has continued ever since, such that nowadays it is used by most contemporary GPOSs, including Linux, the BSD family, Solaris, AIX, HP-UX, IRIX, and the Windows family. Roughly speaking, the way the mechanism works is that at boot-time the kernel sets a hardware clock to generate periodic interrupts at fixed intervals (every few milliseconds; anywhere between 1ms to 15ms, depending on the OS). The time instance at which the interrupt fires is called a *tick*, and the elapsed time between two consecutive ticks is called a *tick duration*. The interrupt invokes a kernel routine, called the *tick handler* that is responsible for various OS activities, of which the following are relevant for our purposes:

1. **Delivering timing services** and alarm signals. For example, a movie player that wants to wakeup on time to display the next frame, requests the OS (using a system call) to wake it up at the designated time. The kernel places this request in an internal data structure that is checked upon each tick. When the tick handler discovers there’s an expired alarm, it wakes the associated player up. The player then displays the frame and the scenario is repeated until the movie ends.
2. **Accounting for CPU usage** by recording that the currently running process  $S$  consumed CPU cycles during the last tick. Specifically, on every tick,  $S$  is stopped, the tick-handler is started, and the kernel increments  $S$ ’s CPU-consumption tick-counter within its internal data structure.
3. **Initiating involuntary preemption** and thereby implementing multitasking (interleaving of the

CPU between several programs to create the illusion they execute concurrently). Specifically, after  $S$  is billed for consuming CPU during the last tick, the tick handler checks whether  $S$  has exhausted its “quantum”, and if so,  $S$  is preempted in favor of another process. Otherwise, it is resumed.

## 1.2 The Concealment Component

The fundamental vulnerability of the tick mechanism lies within the second item above: CPU billing is based on periodic sampling. Consequently, if  $S$  can somehow manage to arrange things such that it always starts to run just after the clock tick, and always goes to sleep just before the next one, then  $S$  will never be billed. One might naively expect this would not be a problem because applications cannot request timing services independent of OS ticks. Indeed, it is *technically impossible* for non-privileged applications to request the OS to deliver alarm signals in between ticks. Nevertheless, we will show that there are several ways to circumvent this difficulty.

To make things even worse, the cheat attack leads to *misaccounting*, where another process is billed for CPU time used by the cheating process. This happens because billing is done in tick units, and so whichever process happens to run while the tick takes place is billed for the *entire* tick duration, even if it only consumed a small fraction of it. As a result, even if the system administrators suspect something, they will suspect the wrong processes. If a cheating process is not visible through system monitoring tools, the only way to notice the attack is by its effect on throughput. The cheater can further disguise its tracks by moderating the amount of CPU it uses so as not to have too great an impact on system performance.

## 1.3 The Hostile Component

The most basic defense one has against malicious programs is knowing what’s going on in the system. Thus, a situation in which a non-privileged application can conceal the fact it makes use of the CPU, constitutes a serious security problem in its own right. However, there is significantly more to cheat attacks than concealment, because CPU accounting is not conducted just for the sake of knowing what’s going on. Rather, this information has a crucial impact on scheduling decisions.

As exemplified in Section 5, the traditional design principle underlying general-purpose scheduling (as opposed to research or special-purpose schemes) is the same: the more CPU cycles used by a process, the lower its priority becomes [15]. This *negative feedback* (running reduces priority to run more) ensures that (1) all processes get a fair share of the CPU, and that (2) processes that do not use the CPU very much — such as

I/O bound processes — enjoy a higher priority for those bursts in which they want it. In fact, the latter is largely what makes text editors responsive to our keystrokes in an overloaded system [14].

The practical meaning of this is that by consistently appearing to consume 0% CPU, an application gains a very high priority. As a consequence, when a cheating process wakes up and becomes runnable (following the scenario depicted in the previous subsection) it usually has a higher priority than that of the currently running process, which is therefore immediately preempted in favor of the cheater. Thus, as argued above, unprivileged concealment capabilities indeed allow an application to monopolize the CPU. However, surprisingly, this is not the whole story. It turns out that even without concealment capabilities it is still sometimes possible for an application to dominate the CPU without superuser privileges, as discussed next.

#### 1.4 The Interactivity Component and the Spectrum of Vulnerability to Cheating

Not all GPOSs are vulnerable to cheat attacks to the same degree. To demonstrate, let us first compare between Linux-2.4 and Linux-2.6. One of the radical differences between the two is the scheduling subsystem, which has been redesigned from scratch and undergone a complete rewrite. A major design goal of the new scheduler was to improve users' experience by attempting to better identify and service interactive processes. In fact, the lead developer of this subsystem argued that "the improvement in the way interactive tasks are handled is actually the change that should be the most noticeable for ordinary users" [3]. Unfortunately, with this improvement also came increased vulnerability to cheat attacks.

In Linux-2.6, a process need not conceal the fact it is using the CPU in order to monopolize it. Instead, it can masquerade as being "interactive", a concept that is tied within Linux-2.6 to the number of times the process voluntarily sleeps [32]. Full details are given in Section 6, but in a nutshell, to our surprise, even after we introduced cycle-accurate CPU accounting to the Linux-2.6 kernel and made the cheating process fully "visible" at all times, the cheater still managed to monopolize the CPU. The reason turned out to be the cheater's many short voluntary sleep-periods while clock ticks take place (as specified in Section 1.2). This, along with Linux-2.6's aggressive preference of "interactive" processes yielded the new weakness.

In contrast, the interactivity weakness is not present in Linux-2.4, because priorities do not reflect any considerations that undermine the aforementioned negative feedback. Specifically, the time remaining until a process exhausts its allocated quantum also serves as its priority,

and so the negative feedback is strictly enforced [36]. Indeed, having Linux-2.4 use accurate accounting information defeats the cheat attack.

The case of Linux 2.4 and 2.6 is not an isolated incident. It is analogous to the case of FreeBSD and the two schedulers it makes available to its users. The default "4BSD" scheduler [5] is vulnerable to cheat attacks due to the sampling nature of CPU accounting, like Linux-2.4. The newer "ULE" scheduler [42] (designated to replace 4BSD) attempts to improve the service provided to interactive processes, and likewise introduces an additional weakness that is similar to that of Linux-2.6. We conclude that there's a genuine (and much needed) intent to make GPOSs do a better job in adequately supporting newer workloads consisting of modern interactive applications such as movie players and games, but that this issue is quite subtle and prone to errors compromising the system (see Section 5.2 for further discussion of why this is the case).

Continuing to survey the OS spectrum, Solaris represents a different kind of vulnerability to cheat attacks. This OS maintains completely accurate CPU accounting (which is not based on sampling) and does *not* suffer from the interactivity weakness that is present in Linux-2.6 and FreeBSD/ULE. Surprisingly, despite this configuration, it is still vulnerable to the hostile component of cheating. The reason is that, while accurate information is maintained by the kernel, the scheduling subsystem does not make use of it (!). Instead, it utilizes the sampling-based information gathered by the periodic tick handler [35]. This would have been acceptable if all applications "played by the rules" (in which case periodic sampling works quite well), but such an assumption is of course not justified. The fact that the developers of the scheduling subsystems did not replace the sampled information with the accurate one, despite its availability, serves as a testament of their lack of awareness to the possibility of cheat attacks.

Similarly to Solaris, Windows XP maintains accurate accounting that is unused by the scheduler, which maintains its own sample-based statistics. But in contrast to Solaris, XP also suffers from the interactivity weakness of Linux 2.6 and ULE. Thus, utilizing the accurate information would have had virtually no effect.

From the seven OS/scheduler pairs we have examined, only Mac OS X was found to be immune from the cheat attack. The reason for this exception, however, is not a better design of the tick mechanism so as to avoid the attack. Rather, it is because Mac OS X uses a different timing mechanism altogether. Similarly to several realtime OSs, Mac OS X uses *one-shot timers* to drive its timing and alarm events [29, 47, 20]. These are hardware interrupts that are set to go off only for specific needs, rather than periodically. With this design, the OS maintains an

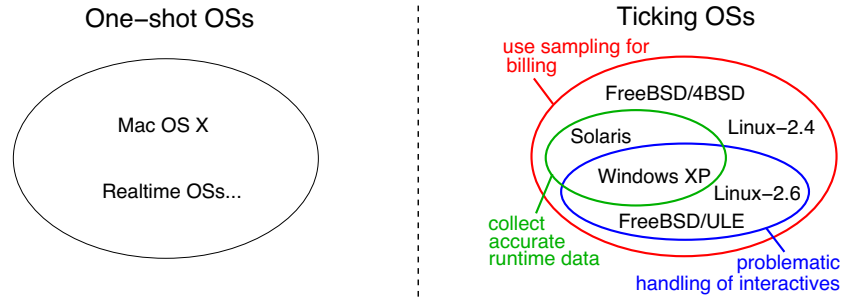


Figure 1: Classification of major operating systems in terms of features relevant for the cheat attack. General-purpose OSs are typically tick-based, in which case they invariably use sampling and are therefore vulnerable to cheat attacks to various degrees.

ascending list of outstanding timers and sets a one-shot event to fire only when it is time to process the event at the head of the list; when this occurs, the head is popped, and a new one-shot event is set according to the new head. This design is motivated by various benefits, such as reduced power consumption in mobile devices [40], better alarm resolution [15], and less OS “noise” [53]. However, it causes the time period between two consecutive timer events to be variable and unbounded. As a consequence, CPU-accounting based on sampling is no longer a viable option, and the Mac OS X immunity to cheat attacks is merely a side effect of this. Our findings regarding the spectrum of vulnerabilities to cheat attacks are summarized in Fig. 1.

While it is possible to rewrite a tick-based OS to be one-shot, this is a non-trivial task requiring a radical change in the kernel (e.g. the Linux-2.6.16 kernel source tree contains 8,997 occurrences of the tick frequency HZ macro, spanning 3,199 files). Worse, ticks have been around for so long, that some user code came to directly rely on them [52]. Luckily, eliminating the threat of cheat attacks does not necessitate a radical change: there exists a much simpler solution (Section 6). Regardless, the root cause of the problem is not implementation difficulties, but rather, lack of awareness.

## 1.5 Roadmap

This paper is structured as follows. Section 2 places the cheat attack within the related context and discusses the potential exploits. Section 3 describes in detail how to implement a cheating process and experimentally evaluates this design. Section 4 further shows how to apply the cheating technique to arbitrary applications, turning them into “cheaters” without changing their source code. Section 5 provides more details on contemporary schedulers and highlights their weaknesses in relation to the cheat attack on an individual basis. Section 6 describes and evaluates our solution to the problem, and Section 7 concludes.

## 2 Potential Exploits and Related Work

### 2.1 The Privileges-Conflict Axis

The conflict between attackers and defenders often revolves around privileges of using resources, notably network, storage, and the CPU. The most aggressive and general manifestation of this conflict is attackers that aspire to have all privileges and avoid all restrictions by obtaining root/administrator access. Once obtained, attackers can make use of all the resources of the compromised machine in an uncontrolled manner. Furthermore, using rootkits, they can do so secretly in order to avoid detection and lengthen the period in which the resources can be exploited. Initially, rootkits simply replaced various system programs, such as `netstat` to conceal network activity, `ls` to conceal files, and `ps/top` to conceal processes and CPU usage [55]. But later rootkits migrated into the kernel [9, 46] and underneath it [27], reflecting the rapid escalation of the concealment/detection battle.

At the other end of the privileges conflict one can find attacks that are more subtle and limited in nature. For example, in order to take control over a single JVM instance running on a machine to which an attacker has no physical access, Govindavajhala and Appel suggest the attacker should “convince it [the machine] to run the [Java] program and then wait for a cosmic ray (or other natural source) to induce a memory error”; they then show that “a single bit error in the Java program’s data space can be exploited to execute arbitrary code with a probability of about 70%” within the JVM instance [21]. When successful, this would provide the attacker with the privileges of the user that spawned the JVM.

When positioning the general vs. limited attacks at opposite ends of the privileges-conflict “axis”, the cheat attack is located somewhere in between. It is certainly not as powerful as having root access and a rootkit, e.g. the attacker cannot manipulate and hide network activity or file usage. On the other hand, the attack is not limited to only one user application, written in a specific language,

on the condition of a low probability event such as a cosmic ray flipping an appropriate bit. Instead, at its fullest, the cheat attack offers non-privileged users one generic functionality of a rootkit: A ubiquitous way to control, manipulate, and exploit one computer resource — CPU cycles — in a fairly secretive manner. In this respect, cheating is analogous to attacks like the one suggested by Borisov et al. that have shown how to circumvent the restrictions imposed by file permissions in a fairly robust way [8]. As with cheating, non-privileged users are offered a generic functionality of rootkits, only this time concerning files. An important difference, however, is that Borisov’s attack necessitates the presence of a root setuid program that uses the `access/open` idiom (a widely discouraged practice [11]<sup>1</sup>), whereas our attack has no requirements but running under a ticking OS.

## 2.2 Denying or Using the Hijacked Cycles

Cheating can obviously be used for launching DoS attacks. Since attackers can hijack any amount of CPU cycles, they can run a program that uselessly consumes e.g. 25%, 50%, or 75% of each tick’s cycles, depending on the extent to which they want to degrade the effective throughput of the system; and with concealment capabilities, users may feel that things work slower, but would be unable to say why. This is similar to “shrew” and “RoQ” (Reduction of Quality) attacks that take advantage of the fact that TCP interprets packet loss as an indication of congestion and halves a connection’s transmission rate in response. With well-timed low-rate DoS traffic patterns, these attacks can throttle TCP flows to a small fraction of their ideal rate while eluding detection [28, 23, 50].

Another related concept is “parasitic computing”, with which one machine forces another to solve a piece of a complex computational problem merely by sending to it malformed IP packets and observing the response [6]. Likewise, instead of just denying the hijacked cycles from other applications, a cheating process can leverage them to engage in actual computation (but in contrast, it can do so effectively, whereas parasitic computing is extremely inefficient). Indeed, Section 4 demonstrates how we secretly monopolized an entire departmental *shared* cluster for our own computational needs, without “doing anything wrong”.

A serious exploit would occur if a cheat application

---

<sup>1</sup>The `access` system call was designed to be used by setuid root programs to check whether the invoking user has appropriate permissions, before opening a respective file. This induces a time-of-check-to-time-of-use (TOCTTOU) race condition whereby an adversary can make a name refer to a different file after the `access` and before the `open`. Thus, its manual page states that “the `access` system call is a potential security hole due to race conditions and should never be used” [1].

was spread using a computer virus or worm. This potential development is very worrying, as it foreshadows a new type of exploit for computer viruses. So far, computer viruses targeting the whole Internet have been used mainly for launching DDoS attacks or spam email [34]. In many cases these viruses and worms were found and uprooted because of their very success, as the load they place on the Internet become unignorable [38]. But Staniford et al. described a “surreptitious” attack by which a worm that requires no special privileges can spread in a much harder to detect contagion fashion, without exhibiting peculiar communication patterns, potentially infecting upwards of 10,000,000 hosts [49]. Combining such worms with our cheat attack can be used to create a concealed ad-hoc supercomputer and run a computational payload on massive resources in minimal time, harvesting a huge infrastructure similar to that amassed by projects like SETI@home [2]. Possible applications include cracking encryptions in a matter of hours or days, running nuclear simulations, and illegally executing a wide range of otherwise regulated computations. While this can be done with real rootkits, the fact it can also potentially be done without ever requiring superuser privileges on the subverted machines is further alarming. Indeed, with methods like Borisov’s (circumvent file permissions [8]), Staniford’s (networked undetected contagion [49]), and ours, one can envision a kind of “rootkit without root privileges”.

## 2.3 The Novelty of Cheating

While the cheat attack is simple, to our knowledge, there is no published record of it, nor any mention of it on the web. Related publications point out that general-purpose CPU accounting might be inaccurate, but never raise the possibility that this can be maliciously exploited. Our first encounter with the attack was, interestingly, when it occurred by chance. While investigating the effect of different tick frequencies [15], we observed that an X server servicing a Xine movie player was only billed for 2% of the cycles it actually consumed, a result of (1) X starting to run just after a tick (following Xine’s repetitive alarm signals to display each subsequent frame, which are delivered by the tick handler), and (2) X finishing the work within 0.8 of a tick duration. This pathology in fact outlined the cheat attack principles. But at the time, we did not realize that this can be maliciously done on purpose.

We were not alone: There have been others that were aware of the accounting problem, but failed to realize the consequences. Liedtke argued that the system/user time-used statistics, as e.g. provided by the `getrusage` system call, might be inaccurate “when short active intervals are timer-scheduled, i.e. start always directly after a clock interrupt and terminate before the next one” [30] (exactly

describing the behavior we observed, but stopping short from recognizing this can be exploited).

The designers of the FreeBSD kernel were also aware this might occur, contending that “since processes tend to synchronize to ‘tick’, the statistics clock needs to be independent to ensure that CPU utilization is correctly accounted” [26]. Indeed, FreeBSD performs the billing activity (second item in Section 1.1) independently of the other tick activities (notably timing), at different times and in a different frequency. But while this design alleviates some of the concerns raised by Liedtke [30] and largely eliminates the behavior we observed [15], it is nonetheless helpless against a cheat attack that factors this design in (Section 5) and only highlights the lack of awareness to the possibility of systematic cheating.

Solaris designers noticed that “CPU usage measurements aren’t as accurate as you may think ... especially at low usage levels”, namely, a process that consumes little CPU “could be sneaking a bite of CPU time whenever the clock interrupt isn’t looking” and thus “appear to use 1% of the system but in fact use 5%” [10]. The billing error was shown to match the inverse of the CPU utilization (which is obviously not the case when cheating, as CPU utilization and the billing error are in fact equal).

Windows XP employs a “partial decay” mechanism, proclaiming that without it “it would be possible for threads never to have their quanta reduced; for example, a thread ran, entered a wait state, ran again, and entered another wait state but was never the currently running thread when the clock interval timer fired” [44]. Like in the FreeBSD case, partial decay is useless against a cheat attack (Section 5), but in contrast, it doesn’t even need to be specifically addressed, reemphasizing the elusiveness of the problem.

We contend, however, that all the above can be considered as anecdotal evidence of the absence of awareness to cheat attacks, considering the bottom line, which is that *all widely used ticking operating systems are susceptible to the attack, and have been that way for years.*<sup>2</sup>

### 3 Implementation and Evaluation

As outlined above, the cheat attack exploits the combination of two operating system mechanisms: periodic sampling to account for CPU usage, and prioritization of processes that use less of the CPU. The idea is to avoid the accounting and then enjoy the resulting high priority. We next detail how the former is achieved.

<sup>2</sup>We conceived the attack a few years after [15], as a result of a dispute between PhD students regarding who gets to use the departmental compute clusters for simulations before some approaching deadlines. We eventually did not exercise the attack to resolve the dispute, except for the experiment described in Section 4.1, which was properly authorized by the system personnel.

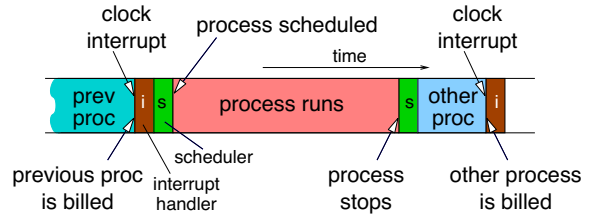


Figure 2: The cheat attack is based on a scenario where a process starts running immediately after one clock tick, but stops before the next tick, so as not to be billed.

#### 3.1 Using the CPU Without Being Billed

When a tick (= periodic hardware clock interrupt) occurs, the entire interval since the previous tick is billed to the application that ran just before the current tick occurred. This mechanism usually provides reasonably accurate billing, despite the coarse tick granularity of a few milliseconds and the fact that nowadays the typical quanta is much shorter, for many applications [15].<sup>3</sup> This is a result of the probabilistic nature of the sampling: Since a large portion of the quanta are shorter than one clock tick, and the scheduler can only count in complete tick units, many of the quanta are not billed at all. But when a short quantum does happen to include a clock interrupt, the associated application is overbilled and charged a full tick. Hence, on average, these two effects tend to cancel out, because the probability that a quantum includes a tick is proportional to its duration.

Fig. 2 outlines how this rationale is circumvented. The depicted scenario has two components: (1) start running after a given billing sample, and (2) stop before the next. Implementing the first component is relatively easy, as both the billing and the firing of pending alarm timers are done upon a tick (first and second items in Section 1.1; handling the situation where the two items are independent, as in FreeBSD, is deferred to Section 5). Consequently, if a process blocks on a timer, it will be released just after a billing sample. And in particular, setting a very short timer interval (e.g. zero nanoseconds) will wake a process up immediately after the very next tick. If in addition it will have high priority, as is the case when the OS believes it is consistently sleeping, it will also start to run.

The harder part is to stop running before the next tick, when the next billing sample occurs. This may happen by chance as described above in relation to Xine and X. The question is how to do this on purpose. Since the OS does not provide intra-tick timing services, the process needs some sort of a finer-grained alternative timing

<sup>3</sup>In this context, quantum is defined to be the duration between the time an application was allocated the CPU and the time in which it relinquished the CPU, either voluntary or due to preemption.

```

inline cycle_t get_cycle

    cycle_t ret
    asm volatile rdtsc : ret
    return ret

cycle_t cycle_per_tic

    nano leep    ero,    // ync with tic
    cycle_t tart    get_cycle

    for int i    i    i
        nano leep    ero,

    return get_cycle    tart /

void cheat_attac    double fraction

    cycle_t wor , tic _ tart, now

    wor    fraction    cycle_per_tic

    nano leep    ero,    // ync with tic
    tic _ tart    get_cycle

    while
        now    get_cycle
        if now    tic _ tart    wor
            nano leep    ero,    // avoid bill
            tic _ tart    get_cycle

        // do ome hort wor here...

```

Figure 3: The complete code for the cheater process (*cycle\_t* is *typedef*-ed to be an unsigned 64-bit integer).

mechanism. This can be constructed with the help of the *cycle counter*, available in all contemporary architectures. The counter is readable from user level using an appropriate assembly instruction, as in the *get\_cycles* function (Fig. 3, top/left) for the Pentium architecture.

The next step is figuring out the interval between each two consecutive clock ticks, in cycles. This can be done by a routine such as *cycles\_per\_tick* (Fig. 3 bottom/left), correctly assuming a zero sleep would wake it up at the next clock interrupt, and averaging the duration of a thousand ticks. While this was sufficient for our purposes, a more precise method would be to tabulate all thousand timestamps individually, calculate the intervals between them, and exclude outliers that indicate some activity interfered with the measurement. Alternatively, the data can be deduced from various OS-specific information sources, e.g. by observing Linux's */proc/interrupts* file (reveals the OS tick frequency) and */proc/cpuinfo* (processor frequency).

It is now possible to write an application that uses any desired fraction of the available CPU cycles, as in the *cheat\_attack* function (Fig. 3, right). This first calculates the number of clock cycles that constitute the desired percentage of the clock tick interval. It then iterates doing its computation, while checking whether the desired limit has been reached at each iteration. When the limit is reached, the application goes to sleep for zero time, blocking till after the next tick. The only assumption is that the computation can be broken into small pieces, which is technically always possible to do (though in Section 4 we further show how to cheat without this assumption). This solves the problem of knowing when to stop to avoid being billed. As a result, this non-privileged application can commandeer any desired percentage of the CPU resources, while looking as if it is using zero resources.

## 3.2 Experimental Results

To demonstrate that this indeed works as described, we implemented such an application and ran it on a 2.8GHz Pentium-IV, running a standard Linux-2.6.16 system default installation with the usual daemons, and no other user processes except our tests. The application didn't do any useful work — it just burned cycles. At the same time we also ran another compute-bound application, that also just burned cycles. An equitable scheduler should have given each about 50% of the machine. But the cheat application was set to use 80%, and got them.

During the execution of the two competing applications, we monitored every important event in the system (such as interrupts and context switches) using the Klogger tool [17]. A detailed rendition of precisely what happened is given in Fig. 4. This shows 10 seconds of execution along the *X* axis, at tick resolution. As the system default tick rate is 250 Hz, each tick represents 4ms. To show what happens during each tick, we spread those 4ms along the *Y* axis, and use color coding. Evidently, the cheat application is nearly always the first one to run (on rare occasions some system daemon runs initially for a short time). But after 3.2ms (that is, exactly 80% of the tick) it blocks, allowing the honest process or some other process to run.

Fig. 5 scatter-plots the billing accuracy, where each point represents one quantum. With accurate accounting we would have seen a diagonal, but this is not the case. While the cheat process runs for just over 3ms each time, it is billed for 0 (bottom right disk). The honest process, on the other hand, typically runs for less than 1ms, but is billed for 4 (top left); on rare occasions it runs for nearly a whole tick, due to some interference that caused the cheater to miss one tick (top right); the cheater nevertheless recovers at the following tick. The other processes run for a very short time and are never billed.

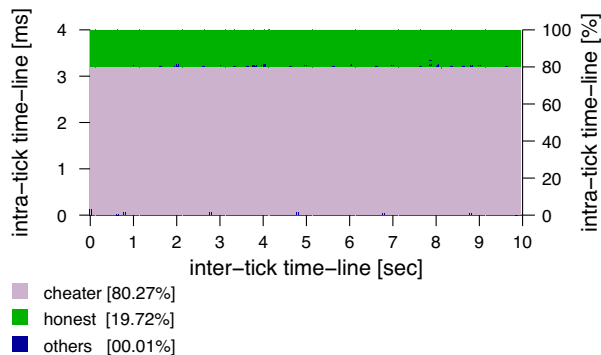


Figure 4: Timeline of 10 seconds of competition between a cheat and honest processes. Legends give the distribution of CPU cycles.

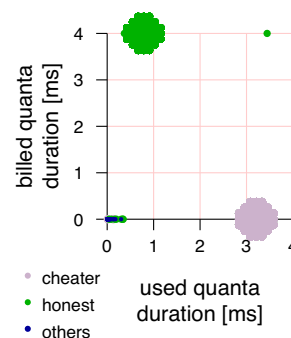


Figure 5: Billing accuracy achieved during the test shown in Fig. 4.

```

a      :      total,      running,      leeping,      topped,      ombie
pu      :      . u ,      . y,      . ni,      . id,      . wa,      . hi,      . i
em:      total,      u ed,      free,      buffer
wap:      total,      u ed,      free,      cached

                                H
dant    21                      R    99.3      .    0:07.79    honest
dant    .                      .    .    .    .    top
dant    .                      .    .    .    .    c h
dant    .                      .    .    .    .    c h
dant    .                      .    .    .    .    bm no log. h
dant    15                      S    0.0      .    0:00.00    cheater

```

Figure 6: Snippet of the output of the `top` utility for user `dants` (the full output includes dozens of processes, and the cheater appears near the end and is hard to notice). The honest process is billed for 99.3% of the CPU, while actually getting only 20%. The cheater looks as if it is not getting any CPU, while it actually consumes 80%.

The poor accounting information propagates to system usage monitoring tools. Like any monitoring utility, the view presented to the user by `top` is based on OS billing information, and presents a completely distorted picture as can be seen in Fig. 6. This dump was taken about 8 seconds into the run, and indeed the honest process is billed for 99.3% of the CPU and is reported as having run for 7.79 seconds. The cheater on the other hand is shown as using 0 time and 0% of the CPU. Moreover, it is reported as being suspended (status S), further throwing off any system administrator that tries to understand what is going on. As a result of the billing errors, the cheater has the highest priority (lowest numerical value: 15), which allows it to continue with its exploits.

Our demonstration used a setting of 80% for the cheat application (a 0.8 fraction argument to the `cheat_attack` function in Fig. 3). But other values can be used. Fig. 7 shows that the attack is indeed very accurate, and can achieve precisely the desired level of usage. Thus, an attacker that wants to keep a low profile can set the cheating to be a relatively small value (e.g. 15%); the chances users will notice this are probably very slim.

Finally, our demonstration have put forth only one competitor against the cheater. But the attack is in

fact successful regardless of the number of competitors that form the background load. This is demonstrated in Fig. 8: An honest process (left) gets its *equal share* of the CPU, which naturally becomes smaller and smaller as more competing processes are added. For example, when 5 processes are present, each gets 20%. In contrast, when the process is cheating (right) it always gets what it wants, despite the growing competition. The reason of course is that the cheater has very a high priority, as it appears as consuming no CPU cycles, which implies an immediate service upon wakeup.

## 4 Running Unmodified Applications

A potential drawback of the above design is that it requires modifying the application to incorporate the cheat code. Ideally, from an attacker’s perspective, there should be a “cheat” utility such that invoking e.g.

```
cheat application
```

would execute the application as a 95%-cheater, without having to modify and recompile its code. This section describes two ways to implement such a tool.

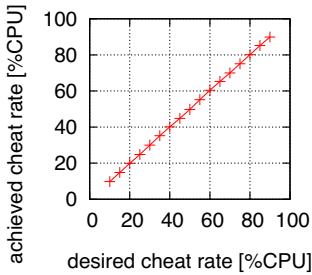


Figure 7: The attack is very accurate and the cheater gets exactly the amount of CPU cycles it requested.

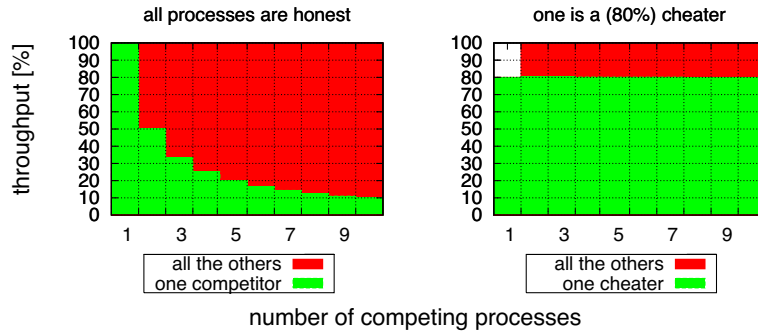


Figure 8: Cheating is immune to the background load.

## 4.1 Cheat Server

The sole challenge a cheat application faces is obtaining a timing facility that is finer than the one provided by the native OS. Any such facility would allow the cheater to systematically block before ticks occur, insuring it is never billed and hence the success of the attack. In the previous section, this was obtained by subdividing the work into short chunks and consulting the cycle counter at the end of each. A possible alternative is obtaining the required service using an external machine, namely, a *cheat server*.

The idea is very simple. Using a predetermined *cheat protocol*, a client opens a connection and requests the remote server to send it a message at some designated high-resolution time, before the next tick on the local host occurs. (The request is a single UDP packet specifying the required interval in nanoseconds; the content of the response message is unimportant.) The client then polls the connection to see if a message arrived, instead of consulting the cycle counter. Upon the message arrival, the client as usual sleeps-zero, wakes up just after the next tick, sends another request to the server, and so on. The only requirement is that the server would indeed be able to provide the appropriate timing granularity. But this can be easily achieved if the server busy-waits on its cycle counter, or if its OS is compiled with a relatively high tick rate (the alternative we preferred).

By switching the fine-grained timing source — from the cycle counter to the network — we gain one important advantage: instead of polling, we can now sleep-wait for the event to occur, e.g. by using the `select` system call. This allows us to divide the cheater into two separate entities: the *target application*, which is the the unmodified program we want to run, and the *cheat client*, which is aware of the cheat protocol, provisions the target application, and makes sure it sleeps while ticks occur. The client exercises its control by using the standard `SIGSTOP/SIGCONT` signals, as depicted in Fig. 9:

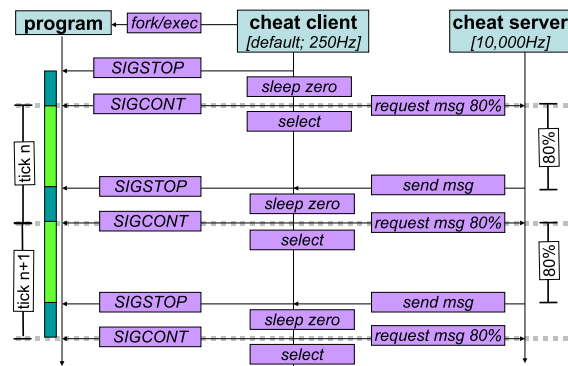


Figure 9: The cheat protocol, as used by a 80%-cheater.

1. The client forks the target application, sends it a **stop** signal, and goes to sleep till the next tick.
2. Awoken on a tick, the client does the following:
  - (a) It sends the cheat server a request for a timing message including the desired interval.
  - (b) It sends the target application a **cont** signal to wake it up.
  - (c) It blocks-waiting for the message from the cheat server to arrive.
3. As the cheat client blocks, the operating system will most probably dispatch the application that was just unblocked (because it looks as if it is always sleeping, and therefore has high priority).
4. At the due time, the cheat server sends its message to the cheat client. This causes a network interrupt, and typically the immediate scheduling of the cheat client (which also looks like a chronic sleeper).
5. The cheat client now does two things:
  - (a) It sends the target application a **stop** signal to prevent it from being billed
  - (b) It goes to sleep-zero, till the next (local) tick.
6. Upon the next tick, it will resume from step 2.

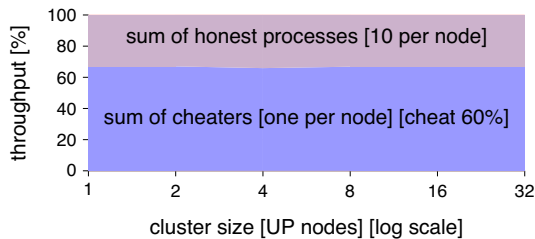


Figure 10: The combined throughput of honest vs. 60%-cheating processes, as a function of the number of cluster nodes used. On each node there are ten honest processes and one cheater running. The cheaters’ throughput indicates that the server simultaneously provides good service to all clients.

To demonstrate that this works, we have implemented this scenario, hijacking a shared departmental cluster of Pentium-IV machines. As a timing server we used an old Pentium-III machine, with a Linux 2.6 system clocked at 10,000 Hz. While such a high clock rate adds overhead [15], this was acceptable since the timing server does not have any other responsibilities. In fact, it could easily generate the required timing messages for the full cluster size, which was 32 cheat clients in our case, as indicated by Fig. 10.

## 4.2 Binary Instrumentation

Using a cheat server is the least wasteful cheating method in terms of throughput loss, as it avoids all polling. The drawback however is the resulting network traffic that can be used to detect the attack, and the network latency which is now a factor to consider (observe the cheaters’ throughput in Fig. 10 that is higher than requested). Additionally, it either requires a dedicated machine to host the server (if it busy-waits to obtain the finer resolution) or the ability to install a new kernel (if resolution is obtained through higher tick rate). Finally, the server constitutes a single point of failure.

Binary instrumentation of the target application is therefore an attractive alternative, potentially providing a way to turn an arbitrary program into a cheater, requiring no recompilation and avoiding the drawbacks of the cheat-server design. The idea is to inject the cheating code directly into the executable, instead of explicitly including it in the source code. To quickly demonstrate the viability of this approach we used *Pin*, a dynamic binary instrumentation infrastructure from Intel [33], primarily used for analysis tasks as profiling and performance evaluation. Being dynamic, *Pin* is similar to a just-in-time (JIT) compiler that allows attaching *analysis routines* to various pieces of the native machine code upon the first time they are executed.

```
void cheat_analy i
    cycle_t c    get_cycle

    if c tic _ tart
        nano leap  ero,
        tic _ tart  get_cycle
```

Figure 11: The injected cheat “analysis” routine. (The *WORK* macro expands to the number of cycles that reflect the desired cheat fraction; the *tick\_start* global variable is initialized beforehand to hold the beginning of the tick in which the application was started.)

The routine we used is listed in Fig. 11. Invoking it often enough would turn any application into a cheater. The question is where exactly to inject this code, and what is the penalty in terms of performance. The answer to both questions is obviously dependent on the instrumented application. For the purpose of this evaluation, we chose to experiment with an event-driven simulator of a supercomputer scheduler we use as the basis of many research effort [39, 19, 18, 51]. Aside from initially reading an input log file (containing a few years worth of parallel jobs’ arrival times, runtimes, etc.), the simulator is strictly CPU intensive. The initial part is not included in our measurements, so as *not* to amortize the instrumentation overhead and overshadow its real cost by hiding it within more expensive I/O operations.

Fig. 12 shows the slowdown that the simulator experienced as a function of the granularity of the injection. In all cases the granularity was fine enough to turn the simulator into a full fledged cheater. Instrumenting every machine instruction in the program incurs a slowdown of 123, which makes sense because this is approximately the duration of *cheat\_analysis* in cycles. This is largely dominated by the *rdtsc* operation (read time-stamp counter; wrapped by *get\_cycles*), which takes about 90 cycles. The next grain size is a *basic block*, namely, a single-entry single-exit instructions sequence (containing no branches). In accordance to the well known assertion that “the average basic block size is around four to five instructions” [56], it incurs a slowdown of 25, which is indeed a fifth of the slowdown associated with instructions. A *trace* of instructions (associated with the hardware trace-cache) is defined to be a single-entry multiple exits sequence of basic blocks that may be separated spatially, but are adjacent temporally [43]. This grain size further reduces the slowdown to 15. Instrumenting at the coarser function level brings us to a slowdown factor of 3.6, which is unfortunately still far from optimal.

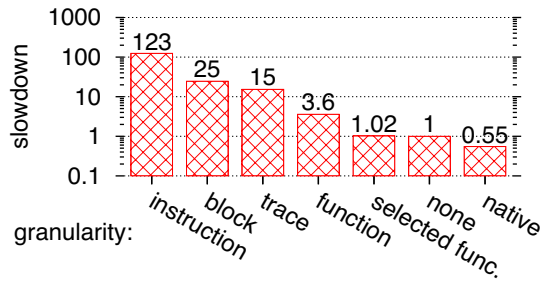


Figure 12: The overheads incurred by cheat-instrumentation, as function of the granularity in which the `cheat_analysis` routine is injected. The Y axis denotes the slowdown penalty due to the instrumentation, relative to the runtime obtained when no instrumentation takes place (“none”).

The average instructions-per-function number within a simulator run, is very small (about 35), the result of multiple abstraction layers within the critical path of execution. This makes the function-granularity inappropriate for injecting the cheat code to our simulator, when attempting to turn it into an efficient cheater. Furthermore, considering the fact that nowadays a single tick consists of millions of cycles (about 11 millions on the platform we used, namely, a 2.8 GHz Pentium-IV at 250 Hz tick rate), a more adequate grain size for the purpose of cheating would be, say, tens of thousands of cycles. Thus, a slightly more sophisticated approach is required. Luckily, simple execution profiling (using Pin or numerous other tools) quickly reveal where an application spends most of its time; in the case of our simulator this was within two functions, the one that pops the next event to simulate and the one that searches for the next parallel job to schedule. By instructing Pin to instrument only these two functions, we were able to turn the simulator into a cheater, while reducing the slowdown penalty to less than 2% of the baseline. We remark that even though this selective instrumentation process required our manual intervention, we believe it reflects a fairly straightforward and simple methodology that can probably be automated with some additional effort.

Finally, note that all slowdowns were computed with respect to the runtime of a simulator that was *not* instrumented, but still executed under Pin. This was done so as not to pollute our evaluation with unrelated Pin-specific performance issues. Indeed, running the simulator natively is 45% faster than the Pin baseline, a result of Pin essentially being a virtual machine [33]. Other binary instrumentation methods, whether static [48], exploiting free space within the executable itself [41], or linking to it loadable modules [4], do not suffer from this deficiency and are expected to close the gap.

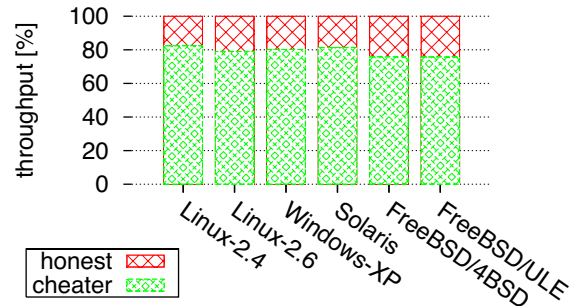


Figure 13: Throughput of a 80%-cheater competing against an honest process, under the operating systems with which we experimented. These measurements were executed on the following OS versions: Linux 2.4.32, Linux 2.6.16, Window XP SP2, Solaris 10 (SunOS 5.10 for i386), and FreeBSD 6.1.

## 5 General-Purpose Schedulers

The results shown so far are associated with Linux-2.6. To generalize, we experimented with other ticking operating systems and found that they are all susceptible to the cheat attack. The attack implementation was usually as shown in Fig. 3, possibly replacing the `nanosleep` with a call to `pause`, which blocks on a repeated tick-resolution alarm-signal that was set beforehand using `setitimer` (all functions are standard POSIX; the exceptions was Windows XP, for which we used Win32’s `GetMessage` for blocking). Fig. 13 shows the outcome of repeating the experiment described in Section 3 (throughput of simultaneously running a 80%-cheater against an honest process) under the various OSs.

Our high level findings were already detailed in Section 1.4 and summarized in Fig. 1. In this section we describe in more detail the design features that make schedulers vulnerable to cheating; importantly, we address the “partial quantum decay” mechanism of Windows XP and provide more details regarding FreeBSD, which separates billing from timing activity and requires a more sophisticated cheating approach.

### 5.1 Multilevel Feedback Queues

Scheduling in all contemporary general-purpose operating systems is based on a *multilevel feedback queue*. The details vary, but roughly speaking, the priority is a combination of a static component (“nice” value), and a dynamic component that mostly reflects lack of CPU usage, interpreted as being “I/O bound”; processes with the highest priority are executed in a round-robin manner. As the cheat process avoids billing, it gets the highest priority and hence can monopolize the CPU. This is what makes cheating widely applicable.

A potential problem with the multilevel feedback queues is that processes with lower priorities might starve. OSs employ various policies to avoid this. For example, **Linux 2.4** uses the notion of “epochs” [36]. Upon a new epoch, the scheduler allocates a new quantum to all processes, namely, allows them to run for an additional 60ms. The epoch will not end until *all runnable processes* have exhausted their allocation, insuring all of them get a chance to run before new allocations are granted. Epochs are initiated by the tick handler, as part of the third item in Section 1.1. The remaining time a process has to run in the current epoch also serves as its priority (higher values imply higher priority). Scheduling decisions are made only when the remaining allocation of the currently running process reaches zero (possibly resulting in a new epoch if no runnable processes with positive allocation exist), or when a blocked process is made runnable.

This design would initially seem to place a limit on the fraction of cycles hijacked by the cheater. However, as always, cheating works because of the manner Linux-2.4 rewards sleepers: upon a new epoch, a currently *blocked* process gets to keep half of its unused allocation, in addition to the default 60ms. As a cheater is never billed and always appears blocked when a tick takes place, its priority quickly becomes  $\sum_{i=0}^{\infty} 60 \cdot 2^{-i} = 120$  (the maximum possible), which means it is always selected to run when it unblocks.

In **Solaris**, the relationship between the priority and the allocated quantum goes the other way [35]. When a thread is inserted to the run-queue, a table is used to allocate its new priority and quantum (which are two separate things here) based on its previous priority and the reason it is inserted into the queue — either because its time quantum expired or because it just woke up after blocking. The table is designed such that processes that consume their entire allocation receive even longer allocations, but are assigned lower priorities. In contrast, threads that block or sleep are allocated higher priorities, but shorter quanta. By avoiding billing the cheater is considered a chronic sleeper that never runs, causing its priority to increase until it reaches the topmost priority available. The short-quanta allocation restriction is circumvented, because the scheduler maintains its own (misguided) CPU-accounting based on sampling ticks.

## 5.2 Prioritization For Interactivity

An obvious feature of the Linux 2.4 and Solaris schemes is that modern interactive processes (as games or movie players that consume a lot of CPU cycles) will end up having low priority and will be delayed as they wait for all other processes to complete their allocations. This is an inherent feature of trying to equally partition the pro-

cessor between competing processes. **Linux 2.6** therefore attempts to provide special treatment to processes it identifies as interactive by maintaining their priority high and by allowing them to continue to execute even if their allocation runs out, provided other non-interactive processes weren’t starved for too long [32]. A similar mechanism is used in the **ULE** scheduler on **FreeBSD** [42].

In both systems, interactive processes are identified based on the ratio between the time they sleep and the time they run, with some weighting of their relative influence. If the ratio passes a certain threshold, the process is deemed interactive. This mechanism plays straight into the hands of the cheater process: as it consistently appears sleeping, it is classified interactive regardless of the specific value of the ratio. The anti-starvation mechanism is irrelevant because other processes are allowed to run at the end of each tick when the cheater sleeps. Thus, cheating would have been applicable even in the face of completely accurate CPU accounting. (The same observation holds for Windows XP, as described in the next subsection.)

We contend that the interactivity weakness manifested by the above is the result of two difficulties. The first is how to identify multimedia applications, which is currently largely based on their observed sleep and CPU consumption patterns. This is a problematic approach: our cheater is an example of a “false positive” it might yield. In a related work we show that this problem is inherent, namely, that typical CPU-intensive interactive application can be paired with non-interactive applications that have identical CPU consumption patterns [14]. Thus, it is probably impossible to differentiate between multimedia applications and others based on such criteria, and attempts to do so are doomed to fail; we argue that the solution lies in directly monitoring how applications interact with devices that are of interest to human users [16].

The second difficulty is how to schedule a process identified as being interactive. The problem here arises from the fact that multimedia applications often have both realtime requirements (of meeting deadlines) and significant computational needs. Such characteristics are incompatible with the negative feedback of “running reduces priority to run more”, which forms the basis of the classic general-purpose scheduling [5] (as in Linux 2.4, Solaris, and FreeBSD/4BSD) and only delivers fast response times to applications that require little CPU. Linux-2.6, FreeBSD/ULE, and Windows XP tackled this problem in a way that compromises the system. And while it is possible to patch this to a certain extent, we argue that any significant divergence from the aforementioned negative-feedback design necessitates a much better notion of what is important to users than can ever be inferred solely from CPU consumption patterns [14, 16].

### 5.3 Partial Quantum Decay

In **Windows XP**, the default time quantum on a workstation or server is 2 or 12 timer ticks, respectively, with the quantum itself having a value of “6” ( $3 \times 2$ ) or “36” ( $3 \times 12$ ), implying that every clock tick decrements the quantum by 3 units [44]. The reason a quantum is stored internally in terms of a multiple of 3 units per tick rather than as single units is to allow for “partial quantum decay”. Specifically, each waiting thread is charged one unit upon wakeup, so as to prevent situations in which a thread avoids billing just because it was asleep when the tick occurred. Hence, the cheater loses a unit upon each tick. Nonetheless, this is nothing but meaningless in comparison to what it gains due its many sleep events.

After a nonzero wait period (regardless of how short), Windows XP grants the awakened thread a “priority boost” by moving it a few steps up within the multi-level feedback queue hierarchy, relative to its base priority. Generally, following a boost, threads are allowed to exhaust their full quantum, after which they are demoted one queue in the hierarchy, allocated another quantum, and so forth until they reach their base priority again. This is sufficient to allow cheating, because a cheater is promoted immediately after being demoted (as it sleeps on every tick). Thus, it consistently maintains a higher position relative to the “non-boosted” threads and therefore always gets the CPU when it awakes. By still allowing others to run at the end of each tick, it prevents the anti-starvation mechanism from kicking in.

Note that this is true regardless of whether the billing is accurate or not, which means XP suffers from the interactivity weakness as Linux 2.6 and FreeBSD/ULE. To make things even worse, “in the case where a wait is not satisfied immediately” (as for cheaters), “its [the thread’s] quantum is reset to a full turn” [44], rendering the partial quantum decay mechanism (as any hypothetical future accurate billing) completely useless.

### 5.4 Dual Clocks

Compromising **FreeBSD**, when configured to use its **4BSD** default scheduler [37], required us to revisit the code given in Fig. 3. Noticing that timer-oriented applications often tend to synchronize with ticks and start to run immediately after they occur, the FreeBSD designers decided to separate the billing from the timing activity [26]. Specifically, FreeBSD uses two timers with relatively prime frequencies — one for interrupts in charge of driving regular kernel timing events (with frequency *HZ*), and one for gathering billing statistics (with frequency *STATHZ*). A running thread’s time quantum is decremented by 1 every *STATHZ* tick. The test sys-

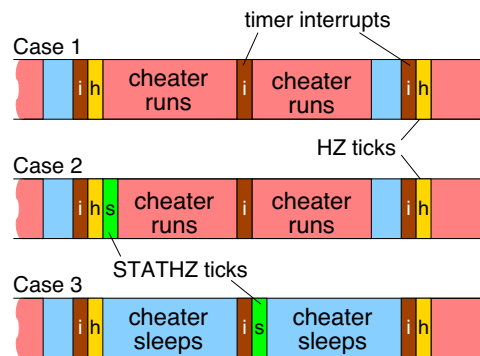


Figure 14: The three possible alignments of the two FreeBSD clocks: no *STATHZ* tick between consecutive *HZ* ticks (case 1), *STATHZ* ticks falls on an even timer interrupt alongside a *HZ* tick (case 2), and a *STATHZ* tick falling on an odd clock interrupt between *HZ* ticks (case 3).

tem that was available to us runs with a *HZ* frequency of 1000Hz and *STATHZ* frequency of  $\sim 133$ Hz.

Both the *HZ* and *STATHZ* timers are derived from a single timer interrupt, configured to fire at a higher frequency of  $2 \times HZ = 2000$ Hz. During each timer interrupt the handler checks whether the *HZ* and/or *STATHZ* tick handlers should be called — the first is called every 2 interrupts, whereas the second is called every 15–16 interrupts ( $\approx \frac{2000}{133}$ ). The possible alignments of the two are shown in Fig. 14. The *HZ* ticks are executed on each even timer interrupt (case 1). Occasionally the *HZ* and *STATHZ* ticks align on an even timer interrupt (case 2), and sometimes *STATHZ* is executed on an odd timer interrupt (case 3). By avoiding *HZ* ticks we also avoid *STATHZ* ticks in case 2. But to completely avoid being billed for the CPU time it consumes, the cheater must identify when case 3 occurs and sleep between the two consecutive *HZ* tick surrounding the *STATHZ* tick.

The kernel’s timer interrupt handler calculates when to call the *HZ* and *STATHZ* ticks in a manner which realigns the two every second. Based on this, we modified the code in Fig. 3 to pre-compute a  $2 \times HZ$  sized *STATHZ*-bitmap, in which each bit corresponds to a specific timer interrupt in a one second interval, and setting the bit for those interrupts which drive a *STATHZ* tick. Further, the code reads the number of timer interrupts that occurred since the system was started, available through a `sysctl` call. The cheater then requests the system for signals at a constant *HZ* rate. The signal handler in turn accesses the *STATHZ* bitmap with a value of  $(interrupt\_index + 1) \bmod (2 \times HZ)$  to check whether the *next* timer interrupt will trigger a *STATHZ* tick. This mechanism allows the cheater thread to identify case 3 and simply sleep until the next *HZ* tick fires. The need to occasionally sleep for two full clock ticks slightly reduces the achievable throughput, as indicated in Fig. 13.

## 6 Protecting Against the Cheat Attack

### 6.1 Degrees of Accuracy

While all ticking OSs utilize information that is exclusively based on sampling for the purpose of scheduling, some operating system also maintain precise CPU-usage information (namely, Solaris and Windows XP). Under this design, each kernel entry/exit is accompanied by reading the cycle counter to make the kernel aware of how many cycles were consumed by the process thus far, as well as to provide the user/kernel usage statistics. (Incidentally this also applies to the one-shot Mac OS X.) Solaris, provides even finer statistics by saving the time a thread spends in each of the thread states (running, blocked, etc.). While such consistently accurate information can indeed be invaluable in various contexts, it does not come without a price.

Consider for example the per system call penalty. Maintaining user/kernel statistics requires that (at least) the following would be added to the system call invocation path: two `rdtsc` operations (of reading the cycle counter at kernel entry and exit), subtracting of the associated values, and adding the difference to some accumulator. On our Pentium-IV 2.8GHz this takes  $\sim 200$  cycles (as each `rdtsc` operation takes  $\sim 90$  cycles and the arithmetics involves 64bit integers on a 32bit machine). This penalty is significant relative to the duration of short system calls, e.g. on our system, `sigprocmask` takes  $\sim 430/1020$  cycles with an invalid/valid argument, respectively, implying 20-47% of added overhead.

Stating a similar case, Liedtke argued against this type of kernel fine-grained accounting [30], and indeed the associated overheads may very well be the reason why systems like Linux and FreeBSD do not provide such a service. It is not our intent to express an opinion on the matter, but rather, to make the tradeoff explicit and to highlight the fact that designers need *not* face it when protecting against cheat attacks. Specifically, there is no need to know *exactly* how many cycles were consumed by a running process upon *each* kernel entry (and user/kernel or finer statistics are obviously irrelevant too). The scheduler would be perfectly happy with a much lazier approach: that the information would be updated *only upon a context switch*. This is a (1) far less frequent and a (2) far more expensive event in comparison to a system call invocation, and therefore the added overhead of reading the cycle counter is made relatively negligible.

### 6.2 Patching the Kernel

We implemented this “lazy” perfect-billing patch within the Linux 2.6.16 kernel. It is only a few dozen lines long. The main modification is in the `task_struct` struc-

ture to replace the `time_slice` field that counts down a process’ CPU allocation in a resolution of “jiffies” (the Linux term for clock ticks). It is replaced by two fields: `ns_time_slice`, which counts down the allocated time slice in nanoseconds instead of jiffies, and `ns_last_update`, which records when `ns_time_slice` was last updated. The value of `ns_time_slice` is decremented by the elapsed time since `ns_last_update`, in two places: on each clock tick (this simply replaces the original `time_slice` jiffy decrement, but with the improvement of only accounting for cycles actually used by this process), and from within the `schedule` function just before a context switch (this is the new part). The rest of the kernel is unmodified, and still works in a resolution of jiffies. This was done by replacing accesses to `time_slice` with an inlined function that wraps `ns_time_slice` and rounds it to jiffies.

Somewhat surprisingly, using this patch did not solve the cheat problem: a cheat process that was trying to obtain 80% of the cycles still managed to get them, despite the fact that the scheduler had full information about this (Fig. 15). As explained in Section 5, this happened because of the extra support for “interactive” processes introduced in the 2.6 kernel. The kernel identifies processes that yield a lot as interactive, provided their `nice` level is not too high. When an “interactive” process exhausts its allocation and should be moved from the “active array” into the “expired array”, it is nevertheless allowed to remain in the active array, as long as already expired processes are not starved (they’re not: the cheater runs less than 100% of the time by definition, and thus the Linux anti-starvation mechanism is useless against it). In effect, the scheduler is overriding its own quanta allocations; this is a good demonstration of the two sides of cheating prevention: it is not enough to have good information — it is also necessary to use it effectively.

In order to rectify the situation, we disallowed processes to circumvent their allocation by commenting out the line that reinserts expired “interactive” processes to the active array. As shown in Fig. 16, this has finally succeeded to defeat the attack. The timeline is effectively divided into epochs of 200ms (corresponding to the combined duration of the two 100ms time-slices of the two competing processes) in which the processes share the CPU equitably. While the “interactive” cheater has higher priority (as its many block events gains it a higher position in the multilevel queue hierarchy), this is limited to the initial part of the epoch, where the cheater repeatedly gets to run first upon each tick. However, after  $\sim 125$ ms of which the cheater consumes 80%, its allocation runs out ( $125\text{ms} \cdot \frac{80}{100} = 100\text{ms}$ ). It is then moved to the expired array and its preferential treatment is temporarily disabled. The honest process is now allowed to catch up and indeed runs for  $\sim 75$ ms until it too exhausts

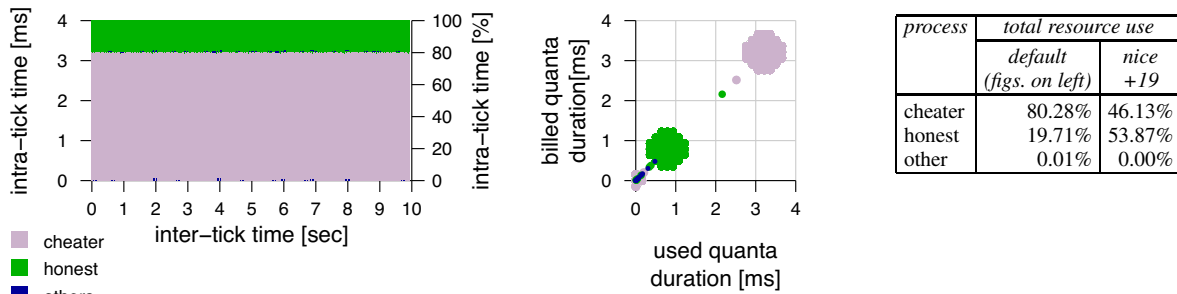


Figure 15: In Linux 2.6, cheating is still possible even with perfect billing (compare with Figs. 4-5).

its quantum and is removed from the active array, leaving it empty. At this point the expired/active array are swapped and the whole thing is repeated.

The above exposes a basic tradeoff inherent to prioritizing based on CPU consumption patterns: one must either enforce a relatively equitable distribution of CPU cycles, or be exposed to attacks by cheaters that can easily emulate “interactive” behavior. (We note in passing that processes with +19 nice value are never regarded as interactive by the 2.6 kernel, so the “optimization” that allows interactive processes to deprive the others is effectively disabled; see right table in Fig. 15.)

Finally, let us discuss the patch overheads. The schedule function was  $\sim 80$  cycles ( $\approx 5\%$ ) slower:  $1636 \pm 182$  cycles on average instead of  $1557 \pm 159$  without the patch ( $\pm$  denotes standard deviation). At the same time, the overhead of a tick handler (the `scheduler_tick` function) was reduced by 17%, from  $8439 \pm 9323$  to  $6971 \pm 9506$ . This is probably due to the fact that after the patch, the cheater ran much less, and therefore generated a lot less timers for the handler to process. Note that these measurements embody the *direct* overhead only (does not include traps to the kernel and back, nor cache pollution due to the traps or context switches). Also note that as the high standard deviations indicate, the distribution of ticks has a long tail, with maximal values around 150,000 cycles. Lastly, the patch did not affect the combined throughput of the processes, at all.

### 6.3 Other Potential Solutions

Several solutions may be used to prevent cheating applications from obtaining excessive CPU resources. Here we detail some of them, and explain why they are inferior to the accurate billing we suggested above. Perhaps the simplest solution is to charge for CPU usage up-front, when a process is scheduled to run, rather than relying on sampling of the running process. However, this will overcharge interactive processes that in fact do not use much CPU time. Another potential solution is to use two clocks, but have the billing clock operate at a finer resolution than the timer clock. This leads to two prob-

lems. One is that it requires a very high tick rate, which leads to excessive overhead. The other is that it does not completely eliminate the cheat attack. An attack is still possible using an extension of the cheat server approach described in Section 4. The extension is that the server is used not only to stop execution, but also to start it. A variant of this is to randomize the clock in order to make it impossible for an attacker to predict when ticks will occur as suggested by Liedtke in relation to user/kernel statistics [30]. This can work, but at the cost of overheads and complexity. Note however that true randomness is hard to come by, and it has already been shown that a system’s random number generator could be reverse-engineered in order to beat the randomness [24]. A third possible approach is to block access to the cycle counter from user level (this is possible at least on the Intel machines). This again suffers from two problems. First, it withdraws a service that may have good and legitimate uses. Second, it too does not eliminate the cheat attack, only make it somewhat less accurate. A cheat application can still be written without access to a cycle counter by finding approximately how much application work can be done between ticks, and using this directly to decide when to stop running.

### 6.4 A Note About Sampling

In the system domain, it is often tempting to say “let us do this chore periodically”. It is simple and easy and therefore often the right thing to do. But if the chore is somehow related to accounting or safeguarding a system, and if “periodically” translates to “can be anticipated”, then the design might be vulnerable. This observation is hardly groundbreaking. However, as with ticks, we suspect it is often brushed aside for the sake of simplicity. Without any proof, we now list a few systems that may possess this vulnerability.

At a finer granularity than ticks, one can find Cisco’s *NetFlow* router tool that “preforms 1 in N periodic [non-probabilistic] sampling” [13] (possibly allowing an adversary to avoid paying for his traffic). At coarser granularity is found the per-node *infod* of the *MOSIX* cluster

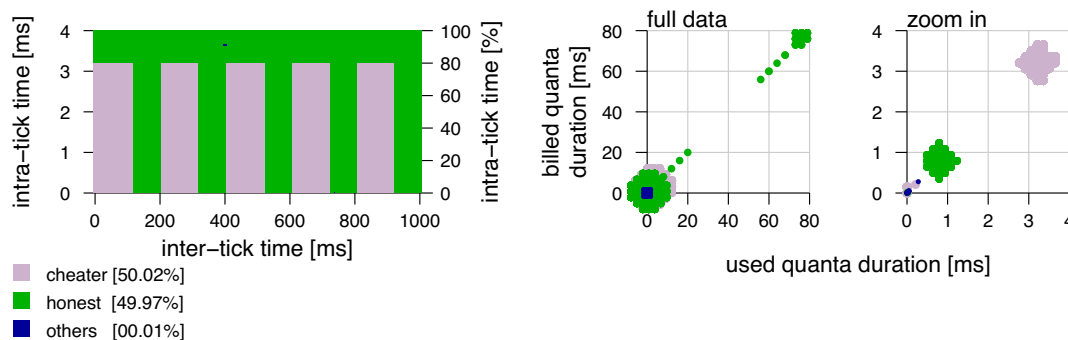


Figure 16: Cheating is eliminated when expired processes are not reinserted to the active list (compare with Fig. 15).

infrastructure [7], which wakes up every 5 seconds to charge processes that migrated to the node (work can be partitioned to shorter processes). The FAQ of IBM's internal file infrastructure called *GSA* (Global Storage Architecture) states that "charges will be based on daily file space snapshots" [22] (raising the possibility of a well-timed *mv* between two malicious cooperating users). And finally, the US Army *MDARS* (Mobile Detection Assessment Response System) patrol robots that "stop periodically during their patrols to scan for intruders using radar and infrared sensors" in search of moving objects [45] again raise the question of what exactly does "periodically" mean.

## 7 Conclusions

The "cheat" attack is a simple way to exploit computer systems. It allows an unprivileged user-level application to seize whatever fraction of the CPU cycles it wants, often in a secretive manner. The cycles used by the cheater are attributed to some other innocent application or simply unaccounted for, making the attack hard to detect. Such capabilities are typically associated with rootkits that, in contrast, require an attacker to obtain superuser privileges. We have shown that all major general-purpose systems are vulnerable to the attack, with the exception of Mac OS X that utilizes one-shot timers to drive its timing mechanism.

Cheating is based on two dominant features of general-purpose systems: that CPU accounting and timer servicing are tied to periodic hardware clock interrupts, and that the scheduler favors processes that exhibit low CPU usage. By systematically sleeping when the interrupts occur, a cheater appears as not consuming CPU and is therefore rewarded with a consistent high priority, which allows it to monopolize the processor.

The first step to protect against the cheat attack is to maintain accurate CPU usage information. This is already done by Solaris and Windows XP that account for each kernel entry. In contrast, by only accounting for

CPU usage before a context switch occurs, we achieve sufficient accuracy in a manner more suitable for systems like Linux and FreeBSD that are unwilling to pay the associated overhead of the Solaris/Windows way. Once the information is available, the second part of the solution is to *incorporate* it within the scheduling subsystem (Solaris and XP don't do that).

The third component is to *use the information judiciously*. This is not an easy task, as indicated by the failure of Windows XP, Linux 2.6, and FreeBSD/ULE to do so, allowing a cheater to monopolize the CPU regardless of whether accurate information is used for scheduling or not. In an attempt to better support the ever increasing CPU-intensive multimedia component within the desktop workload, these systems have shifted to prioritizing processes based on their sleep-events *frequency*, instead of *duration*. This major departure from the traditional general-purpose scheduler design [5] plays straight into the hands of cheaters, which can easily emulate CPU-usage patterns that multimedia applications exhibit. A safer alternative would be to explicitly track user interactions [14, 16].

## Acknowledgments

Many thanks are due to Tal Rabin, Douglas Lee Schales, and Wietse Venema for providing feedback and helping to track down the Tsutomu Shimomura connection.

## References

- [1] *access(2) manual*, *FreeBSD*. URL <http://www.freebsd.org/cgi/man.cgi?query=access>.
- [2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETIhome: an experiment in public-resource computing". *Comm. of the ACM (CACM)* **45(11)**, pp. 56–61, Nov 2002.
- [3] J. Andrews, "Interview: Ingo Molnar". URL <http://kerneltrap.org/node/517>, Dec 2002.

- [4] A. Anonymous, "Building ptrace injecting shell-codes". *Phrack* **10(59)**, p. 8, Jul 2002. URL <http://www.phrack.org/archives/59>.
- [5] M. J. Bach, *The Design of the UNIX Operating System*. Prentice-Hall, 1986.
- [6] A-L. Barabasi, V. W. Freeh, H. Jeong, and J. B. Brockman, "Parasitic computing". *Nature* **412**, pp. 894–897, Aug 2001.
- [7] A. Barak, A. Shiloh, and L. Amar, "An organizational grid of federated MOSIX clusters". In *5th IEEE Int'l Symp. on Cluster Comput. & the Grid (CCGrid)*, May 2005.
- [8] N. Borisov, R. Johnson, N. Sastry, and D. Wagner, "Fixing races for fun and profit: how to abuse atime". In *14th USENIX Security Symp.*, pp. 303–314, Jul 2005.
- [9] J. Butler, J. Undercoffer, and J. Pinkston, "Hidden processes: the implication for intrusion detection". In *IEEE-Information Assurance Workshop (IAW)*, pp. 116–121, Jun 2003.
- [10] A. Cockcroft, "How busy is the CPU, really?". *SunWorld* **12(6)**, Jun 1998.
- [11] D. Dean and A. J. Hu, "Fixing races for fun and profit: how to use access(2)". In *13th USENIX Security Symp.*, pp. 195–206, Aug 2004.
- [12] E. W. Dijkstra, "The structure of the "THE"-multiprogramming system". *Comm. of the ACM (CACM)* **11(5)**, pp. 341–346, May 1968.
- [13] C. Estan and G. Varghese, "New directions in traffic measurements and accounting: focusing on the elephants, ignoring the mice". *ACM Trans. Comput. Syst.* **21(3)**, pp. 270–313, Aug 2003.
- [14] Y. Etsion, D. Tsafirir, and D. G. Feitelson, "Desktop scheduling: how can we know what the user wants?". In *14th Int'l Workshop on Network & Operating Syst. Support or Digital Audio & Video (NOSSDAV)*, pp. 110–115, Jun 2004.
- [15] Y. Etsion, D. Tsafirir, and D. G. Feitelson, "Effects of clock resolution on the scheduling of interactive and soft real-time processes". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 172–183, Jun 2003.
- [16] Y. Etsion, D. Tsafirir, and D. G. Feitelson, "Process prioritization using output production: scheduling for multimedia". *ACM Trans. on Multimedia Comput. Commun. & Appl. (TOMCCAP)* **2(4)**, pp. 318–342, Nov 2006.
- [17] Y. Etsion, D. Tsafirir, S. Kirkpatrick, and D. G. Feitelson, "Fine grained kernel logging with Klogger: experience and insights". In *ACM EuroSys*, Mar 2007.
- [18] D. G. Feitelson, "Experimental analysis of the root causes of performance evaluation results: a backfilling case study". *IEEE Trans. on Parallel & Distributed Syst. (TPDS)* **16(2)**, pp. 175–182, Feb 2005.
- [19] D. G. Feitelson, "Metric and workload effects on computer systems evaluation". *Computer* **36(9)**, pp. 18–25, Sep 2003.
- [20] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz, "The pebble component-based operating system". In *USENIX Annual Technical Conference*, June 1999.
- [21] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine". In *IEEE Symp. on Security and Privacy (SP)*, pp. 154–165, May 2003.
- [22] "Global storage architecture (GSA): FAQs". URL [http://pokgsa.ibm.com/projects/ccgsa/docs/gsa\\_faqs.shtml](http://pokgsa.ibm.com/projects/ccgsa/docs/gsa_faqs.shtml). IBM internal document.
- [23] M. Guirguis, A. Bestavros, and I. Matta, "Exploiting the transients of adaptation for RoQ attacks on internet resources". In *12th IEEE Int'l Conf. on Network Protocols (ICNP)*, pp. 184–195, Oct 2004.
- [24] Z. Gutterman and D. Malkhi, "Hold your sessions: an attack on java servlet session-id generation". In *Topics in Cryptology — CT-RSA 2005*, A. Menezes (ed.), pp. 44–57, Springer-Verlag, Feb 2005. *Lect. Notes Comput. Sci.* vol. 3376.
- [25] G. Hoglund and J. Butler, *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2005.
- [26] P. Jeremy et al., "CPU-timer rate". Dec 2005. Thread from the "freebsd-stable — Production branch of FreeBSD source code" mailing list. URL <http://lists.freebsd.org/pipermail/freebsd-stable/2005-December/020386.html>.
- [27] S. T. King, P. M. Chen, C. V. Yi-Min Wang, H. J. Wang, and J. R. Lorch, "SubVirt: implementing malware with virtual machines". In *IEEE Symp. on Security and Privacy (SP)*, p. 14, May 2006.
- [28] A. Kuzmanovic and E. W. Knightly, "Low-rate TCP-targeted denial of service attacks (the shrew vs. the mice and elephants)". In *ACM SIGCOMM Conf. on Appl. Technologies Archit. & Protocols for Comput. Commun.*, pp. 75–86, Aug 2003.
- [29] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The design and implementation of an operating system to support distributed multimedia applications". *IEEE J. Select Areas in Commun.* **14(7)**, pp. 1280–1297, Sep 1996.
- [30] J. Liedtke, "A short note on cheap fine-grained time measurement". *ACM Operating Syst. Review (OSR)* **30(2)**, pp. 92–94, Apr 1996.
- [31] U. Lindqvist and E. Jonsson, "How to systematically classify computer security intrusions". In *IEEE Symp. on Security and Privacy (SP)*, pp. 154–163, May 1997.
- [32] R. Love, *Linux Kernel Development*. Novell Press, 2nd ed., 2005.
- [33] C-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation". In *ACM Int'l Conf. on Programming Lang. Design & Impl. (PLDI)*, pp. 190–200, Jun 2005. Site: [rogue.colorado.edu/Pin](http://rogue.colorado.edu/Pin).

- [34] J. Markoff, "Attack of the zombie computers is growing threat". *New York Times*, Jan 2007.
- [35] J. Mauro and R. McDougall, *Solaris Internals*. Prentice Hall, 2001.
- [36] S. Maxwell, *Linux Core Kernel Commentary*. Coriolis Group Books, 2nd ed., 2001.
- [37] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.
- [38] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm". *IEEE Security & Privacy (S&P)* **1(4)**, pp. 33–38, Jul/Aug 2003.
- [39] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling". *IEEE Trans. Parallel & Distributed Syst.* **12(6)**, pp. 529–543, Jun 2001.
- [40] C. M. Olsen and C. Narayanaswami, "PowerNap: An efficient power management scheme for mobile devices". *IEEE Trans. on Mobile Computing* **5(7)**, pp. 816–828, Jul 2006.
- [41] P. Padala, "Playing with ptrace, part II". *Linux J.* **2002(104)**, p. 4, Dec 2002. URL <http://www.linuxjournal.com/article/6210>.
- [42] J. Roberson, "ULE: a modern scheduler for FreeBSD". In *USENIX BSDCon*, pp. 17–28, Sep 2003.
- [43] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching". In *29th IEEE Int'l Symp. on Microarchit. (MICRO)*, pp. 24–35, 1996.
- [44] M. E. Russinovich and D. A. Solomon, *Microsoft Windows Internals*. Microsoft Press, 4th ed., Dec 2004.
- [45] B. Shoop, D. M. Jaffee, and R. Laird, "Robotic guards protect munitions". *Army AL&T*, p. 62, Oct-Dec 2006.
- [46] S. Sparks and J. Butler, "Shadow walker: raising the bar for windows rootkit detection". *Phrack* **11(63)**, p. 8, Jul 2005. URL <http://www.phrack.org/archives/63>.
- [47] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus, "A firm real-time system implementation using commercial off-the-shelf hardware and free software". In *4th IEEE Real-Time Technology & App. Symp.*, pp. 112–119, Jun 1998.
- [48] A. Srivastava and A. Eustace, "ATOM: a system for building customized program analysis tools". *SIGPLAN Notices (Best of PLDI 1979-1999)* **39(4)**, pp. 528–539, Apr 2004.
- [49] S. Staniford, V. Paxson, and N. Weaver, "How to own the internet in your spare time". In *11th USENIX Security Symp.*, pp. 149–167, Aug 2002.
- [50] H. Sun, J. C. S. Lui, and D. K. Y. Yau, "Distributed mechanism in detecting and defending against the low-rate TCP attack". *Comput. Networks* **50(13)**, p. Sep, 2006.
- [51] D. Tsafirir, Y. Etsion, and D. G. Feitelson, "Backfilling using system-generated predictions rather than user runtime estimates". *IEEE Trans. on Parallel & Distributed Syst. (TPDS)*, 2007. To appear.
- [52] D. Tsafirir, Y. Etsion, and D. G. Feitelson, *General-Purpose Timing: The Failure of Periodic Timers*. Technical Report 2005-6, Hebrew University, Feb 2005.
- [53] D. Tsafirir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, OS clock ticks, and fine-grained parallel applications". In *19th ACM Int'l Conf. on Supercomput. (ICS)*, pp. 303–312, Jun 2005.
- [54] W. Z. Venema, D. L. Schales, and T. Shimomura, "The novelty and origin of the cheat attack". Jan 2007. Private email communication.
- [55] K. M. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Shermann, and K. A. Oostendorp, "Confining root programs with domain and type enforcement (DTE)". In *6th USENIX Security Symp.*, Jul 1996.
- [56] T-Y. Yeh and Y. N. Patt, "Branch history table indexing to prevent pipeline bubbles in wide-issue superscalar processors". In *26th IEEE Int'l Symp. on Microarchit. (MICRO)*, pp. 164–175, May 1993.

# Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems

Thomas Moscibroda   Onur Mutlu  
Microsoft Research  
{moscitho,onur}@microsoft.com

## Abstract

We are entering the multi-core era in computer science. All major high-performance processor manufacturers have integrated at least two cores (processors) on the same chip — and it is predicted that chips with many more cores will become widespread in the near future. As cores on the same chip share the DRAM memory system, multiple programs executing on different cores can interfere with each others' memory access requests, thereby adversely affecting one another's performance.

In this paper, we demonstrate that current multi-core processors are vulnerable to a new class of Denial of Service (DoS) attacks because the memory system is “unfairly” shared among multiple cores. An application can maliciously destroy the memory-related performance of another application running on the same chip. We call such an application a *memory performance hog (MPH)*. With the widespread deployment of multi-core systems in commodity desktop and laptop computers, we expect MPHs to become a prevalent security issue that could affect almost all computer users.

We show that an MPH can reduce the performance of another application by 2.9 times in an existing dual-core system, without being significantly slowed down itself; and this problem will become more severe as more cores are integrated on the same chip. Our analysis identifies the root causes of unfairness in the design of the memory system that make multi-core processors vulnerable to MPHs. As a solution to mitigate the performance impact of MPHs, we propose a new memory system architecture that provides fairness to different applications running on the same chip. Our evaluations show that this memory system architecture is able to effectively contain the negative performance impact of MPHs in not only dual-core but also 4-core and 8-core systems.

## 1 Introduction

For many decades, the performance of processors has increased by hardware enhancements (increases in clock frequency and smarter structures) that improved single-thread (sequential) performance. In recent years, however, the immense complexity of processors as well as limits on power-consumption has made it increasingly difficult to further enhance single-thread performance [18]. For this reason, there has been a paradigm

shift away from implementing such additional enhancements. Instead, processor manufacturers have moved on to integrating multiple processors on the same chip in a tiled fashion to increase system performance power-efficiently. In a *multi-core chip*, different applications can be executed on different processing cores concurrently, thereby improving overall system throughput (with the hope that the execution of an application on one core does not interfere with an application on another core). Current high-performance general-purpose computers have at least two processors on the same chip (e.g. Intel Pentium D and Core Duo (2 processors), Intel Core-2 Quad (4), Intel Montecito (2), AMD Opteron (2), Sun Niagara (8), IBM Power 4/5 (2)). And, the industry trend is toward integrating many more cores on the same chip. In fact, Intel has announced experimental designs with up to 80 cores on chip [16].

The arrival of multi-core architectures creates significant challenges in the fields of computer architecture, software engineering for parallelizing applications, and operating systems. In this paper, we show that there are important challenges beyond these areas. In particular, we expose a new security problem that arises due to the design of multi-core architectures – a Denial-of-Service (DoS) attack that was not possible in a traditional single-threaded processor.<sup>1</sup> We identify the “security holes” in the hardware design of multi-core systems that make such attacks possible and propose a solution that mitigates the problem.

In a multi-core chip, the DRAM memory system is shared among the threads concurrently executing on different processing cores. The way current DRAM memory systems work, it is possible that a thread with a particular memory access pattern can occupy shared resources in the memory system, preventing other threads from using those resources efficiently. In effect, the

---

<sup>1</sup>While this problem could also exist in SMP (symmetric shared-memory multiprocessor) and SMT (simultaneous multithreading) systems, it will become much more prevalent in multi-core architectures which will be widely deployed in commodity desktop, laptop, and server computers.

memory requests of some threads can be denied service by the memory system for long periods of time. Thus, an aggressive memory-intensive application can severely degrade the performance of other threads with which it is co-scheduled (often without even being significantly slowed down itself). We call such an aggressive application a *Memory Performance Hog (MPH)*. For example, we found that on an existing dual-core Intel Pentium D system one aggressive application can slow down another co-scheduled application by 2.9X while it suffers a slowdown of only 18% itself. In a simulated 16-core system, the effect is significantly worse: the same application can slow down other co-scheduled applications by 14.6X while it slows down by only 4.4X. This shows that, although already severe today, the problem caused by MPHs will become much more severe as processor manufacturers integrate more cores on the same chip in the future.

There are three discomfiting aspects of this novel security threat:

- First, an MPH can maliciously destroy the memory-related performance of other programs that run on different processors on the same chip. Such Denial of Service in a multi-core memory system can ultimately cause significant discomfort and productivity loss to the end user, and it can have unforeseen consequences. For instance, an MPH (perhaps written by a competitor organization) could be used to fool computer users into believing that some other applications are inherently slow, even without causing easily observable effects on system performance measures such as CPU usage. Or, an MPH can result in very unfair billing procedures on grid-like computing systems where users are charged based on CPU hours [9].<sup>2</sup> With the widespread deployment of multi-core systems in commodity desktop, laptop, and server computers, we expect MPHs to become a much more prevalent security issue that could affect almost all computer users.
- Second, the problem of memory performance attacks is radically different from other, known attacks on shared resources in systems, because it cannot be prevented in software. The operating system or the compiler (or any other application) has no direct control over the way memory requests are scheduled in the DRAM memory system. For this reason, even carefully designed and otherwise highly secured systems are vulnerable to memory performance attacks, unless a solution is implemented in *memory system*

<sup>2</sup>In fact, in such systems, some users might be tempted to rewrite their programs to resemble MPHs so that they get better performance for the price they are charged. This, in turn, would unfairly slow down co-scheduled programs of other users and cause other users to pay much higher since their programs would now take more CPU hours.

*hardware* itself. For example, numerous sophisticated software-based solutions are known to prevent DoS and other attacks involving mobile or untrusted code (e.g. [10, 25, 27, 5, 7]), but these are unsuited to prevent our memory performance attacks.

- Third, while an MPH can be designed intentionally, a regular application can unintentionally behave like an MPH and damage the memory-related performance of co-scheduled applications, too. This is discomfiting because an existing application that runs without significantly affecting the performance of other applications in a single-threaded system may deny memory system service to co-scheduled applications in a multi-core system. Consequently, critical applications can experience severe performance degradations if they are co-scheduled with a non-critical but memory-intensive application.

The fundamental reason why an MPH can deny memory system service to other applications lies in the “unfairness” in the design of the multi-core memory system. State-of-the-art DRAM memory systems service memory requests on a First-Ready First-Come-First-Serve (FR-FCFS) basis to maximize memory bandwidth utilization [30, 29, 23]. This scheduling approach is suitable when a single thread is accessing the memory system because it maximizes the utilization of memory bandwidth and is therefore likely to ensure fast progress in the single-threaded processing core. However, when multiple threads are accessing the memory system, servicing the requests in an order that ignores which thread generated the request can unfairly delay some thread’s memory requests while giving unfair preference to others. As a consequence, the progress of an application running on one core can be significantly hindered by an application executed on another.

In this paper, we identify the causes of unfairness in the DRAM memory system that can result in DoS attacks by MPHs. We show how MPHs can be implemented and quantify the performance loss of applications due to unfairness in the memory system. Finally, we propose a new memory system design that is based on a novel definition of *DRAM fairness*. This design provides memory access fairness across different threads in multi-core systems and thereby mitigates the impact caused by a memory performance hog.

The major contributions we make in this paper are:

- We expose a new Denial of Service attack that can significantly degrade application performance on multi-core systems and we introduce the concept of Memory Performance Hogs (MPHs). An MPH is an application that can destroy the memory-related performance of another application running on a different processing core on the same chip.

- We demonstrate that MPHs are a real problem by evaluating the performance impact of DoS attacks on both real and simulated multi-core systems.
- We identify the major causes in the design of the DRAM memory system that result in DoS attacks: hardware algorithms that are unfair across different threads accessing the memory system.
- We describe and evaluate a new memory system design that provides fairness across different threads and mitigates the large negative performance impact of MPHs.

## 2 Background

We begin by providing a brief background on multi-core architectures and modern DRAM memory systems. Throughout the section, we abstract away many details in order to give just enough information necessary to understand how the design of existing memory systems could lend itself to denial of service attacks by explicitly-malicious programs or real applications. Interested readers can find more details in [30, 8, 41].

### 2.1 Multi-Core Architectures

Figure 1 shows the high-level architecture of a processing system with one core (single-core), two cores (dual-core) and N cores (N-core). In our terminology, a “core” includes the instruction processing pipelines (integer and floating-point), instruction execution units, and the L1 instruction and data caches. Many general-purpose computers manufactured today look like the dual-core system in that they have two separate but identical cores. In some systems (AMD Athlon/Turion/Opteron, Intel Pentium-D), each core has its own private L2 cache, while in others (Intel Core Duo, IBM Power 4/5) the L2 cache is shared between different cores. The choice of a shared vs. non-shared L2 cache affects the performance of the system [19, 14] and a shared cache can be a possible source of vulnerability to DoS attacks. However, this is not the focus of our paper because DoS attacks at the L2 cache level can be easily prevented by providing a private L2 cache to each core (as already employed by some current systems) or by providing “quotas” for each core in a shared L2 cache [28].

Regardless of whether or not the L2 cache is shared, the DRAM Memory System of current multi-core systems is shared among all cores. In contrast to the L2 cache, assigning a private DRAM memory system to each core would significantly change the programming model of shared-memory multiprocessing, which is commonly used in commercial applications. Furthermore, in a multi-core system, partitioning the DRAM memory system across cores (while maintaining a shared-memory programming model) is also undesirable because:

1. DRAM memory is still a very expensive resource in modern systems. Partitioning it requires more DRAM chips along with a separate memory controller for each core, which significantly increases the cost of a commodity general-purpose system, especially in future systems that will incorporate tens of cores on chip.
2. In a partitioned DRAM system, a processor accessing a memory location needs to issue a request to the DRAM partition that contains the data for that location. This incurs additional latency and a communication network to access another processor’s DRAM if the accessed address happens to reside in that partition.

For these reasons, we assume in this paper that each core has a private L2 cache but all cores share the DRAM memory system. We now describe the design of the DRAM memory system in state-of-the-art systems.

### 2.2 DRAM Memory Systems

A DRAM memory system consists of three major components: (1) the DRAM banks that store the actual data, (2) the DRAM controller (scheduler) that schedules commands to read/write data from/to the DRAM banks, and (3) DRAM address/data/command buses that connect the DRAM banks and the DRAM controller.

#### 2.2.1 DRAM Banks

A DRAM memory system is organized into multiple banks such that memory requests to different banks can be serviced in parallel. As shown in Figure 2 (left), each DRAM bank has a two-dimensional structure, consisting of multiple rows and columns. Consecutive addresses in memory are located in consecutive columns in the same row.<sup>3</sup> The size of a row varies, but it is usually between 1-8Kbytes in commodity DRAMs. In other words, in a system with 32-byte L2 cache blocks, a row contains 32-256 L2 cache blocks.

Each bank has one *row-buffer* and data can only be read from this buffer. The row-buffer contains at most a single row at any given time. Due to the existence of the row-buffer, modern DRAMs are not truly random access (equal access time to all locations in the memory array). Instead, depending on the access pattern to a bank, a DRAM access can fall into one of the three following categories:

1. **Row hit:** The access is to the row that is already in the row-buffer. The requested column can simply be read from or written into the row-buffer (called a *column access*). This case results in the lowest latency (typically 30-50ns round trip in commodity

<sup>3</sup>Note that consecutive memory rows are located in different banks.

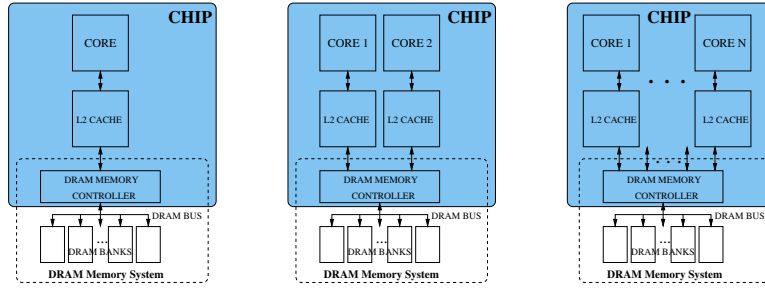


Figure 1: High-level architecture of an example single-core system (left), a dual-core system (middle), and an N-core system (right). The chip is shaded. The DRAM memory system, part of which is off chip, is encircled.

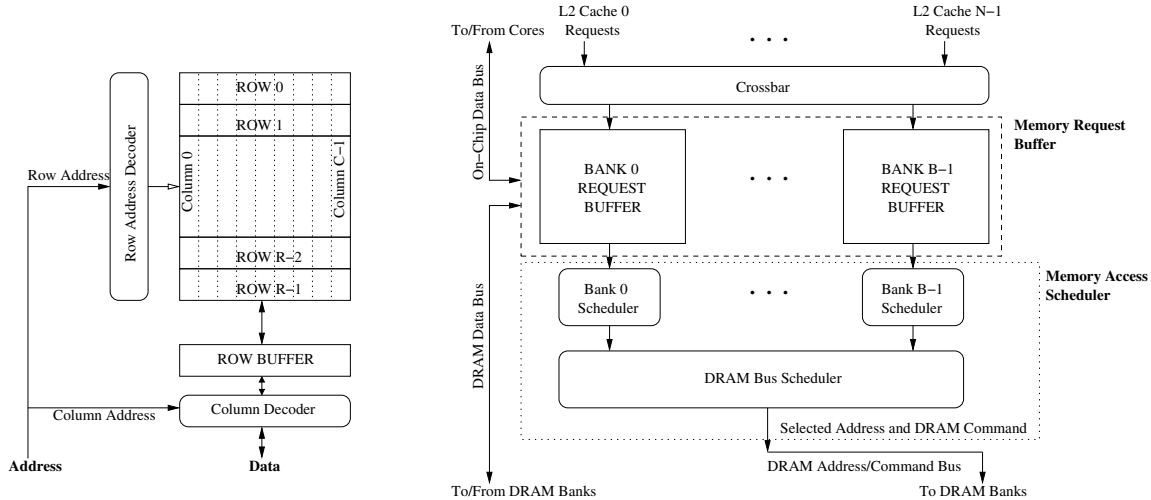


Figure 2: Left: Organization of a DRAM bank, Right: Organization of the DRAM controller

DRAM, including data transfer time, which translates into 90-150 processor cycles for a core running at 3GHz clock frequency). Note that sequential/streaming memory access patterns (e.g. accesses to cache blocks A, A+1, A+2, ...) result in row hits since the accessed cache blocks are in consecutive columns in a row. Such requests can therefore be handled relatively quickly.

2. **Row conflict:** The access is to a row different from the one that is currently in the row-buffer. In this case, the row in the row-buffer first needs to be written back into the memory array (called a *row-close*) because the row access had destroyed the row's data in the memory array. Then, a *row access* is performed to load the requested row into the row-buffer. Finally, a column access is performed. Note that this case has much higher latency than a row hit (typically 60-100ns or 180-300 processor cycles at 3GHz).
3. **Row closed:** There is no row in the row-buffer. Due to various reasons (e.g. to save energy), DRAM memory controllers sometimes close an open row in the row-buffer, leaving the row-buffer empty. In this case, the required row needs to be first loaded into the row-buffer (called a *row access*). Then, a column access is performed. We mention this third case for the

sake of completeness because in the paper, we focus primarily on row hits and row conflicts, which have the largest impact on our results.

Due to the nature of DRAM bank organization, sequential accesses to the same row in the bank have low latency and can be serviced at a faster rate. However, sequential accesses to different rows in the same bank result in high latency. Therefore, to maximize bandwidth, current DRAM controllers schedule accesses to the same row in a bank before scheduling the accesses to a different row even if those were generated earlier in time. We will later show how this policy causes unfairness in the DRAM system and makes the system vulnerable to DoS attacks.

### 2.2.2 DRAM Controller

The DRAM controller is the mediator between the on-chip caches and the off-chip DRAM memory. It receives read/write requests from L2 caches. The addresses of these requests are at the granularity of the L2 cache block. Figure 2 (right) shows the architecture of the DRAM controller. The main components of the controller are the *memory request buffer* and the *memory access scheduler*.

The memory request buffer buffers the requests received for each bank. It consists of separate *bank request*

*buffers*. Each entry in a bank request buffer contains the address (row and column), the type (read or write), the timestamp, and the state of the request along with storage for the data associated with the request.

The memory access scheduler is the brain of the memory controller. Its main function is to select a memory request from the memory request buffer to be sent to DRAM memory. It has a two-level hierarchical organization as shown in Figure 2. The first level consists of separate per-bank schedulers. Each bank scheduler keeps track of the state of the bank and selects the highest-priority request from its bank request buffer. The second level consists of an across-bank scheduler that selects the highest-priority request among all the requests selected by the bank schedulers. When a request is scheduled by the memory access scheduler, its state is updated in the bank request buffer, and it is removed from the buffer when the request is served by the bank (For simplicity, these control paths are not shown in Figure 2).

### 2.2.3 Memory Access Scheduling Algorithm

Current memory access schedulers are designed to maximize the bandwidth obtained from the DRAM memory. As shown in [30], a simple request scheduling algorithm that serves requests based on a first-come-first-serve policy is prohibitive, because it incurs a large number of row conflicts. Instead, current memory access schedulers usually employ what is called a First-Ready First-Come-First-Serve (FR-FCFS) algorithm to select which request should be scheduled next [30, 23]. This algorithm prioritizes requests in the following order in a bank:

1. **Row-hit-first:** A bank scheduler gives higher priority to the requests that would be serviced faster. In other words, a request that would result in a *row hit* is prioritized over one that would cause a *row conflict*.
2. **Oldest-within-bank-first:** A bank scheduler gives higher priority to the request that arrived earliest.

Selection from the requests chosen by the bank schedulers is done as follows:

**Oldest-across-banks-first:** The across-bank DRAM bus scheduler selects the request with the earliest arrival time among all the requests selected by individual bank schedulers.

In summary, this algorithm strives to maximize DRAM bandwidth by scheduling accesses that cause row hits first (regardless of when these requests have arrived) within a bank. Hence, streaming memory access patterns are prioritized within the memory system. The oldest row-hit request has the highest priority in the memory access scheduler. In contrast, the youngest row-conflict request has the lowest priority.

## 2.3 Vulnerability of the Multi-Core DRAM Memory System to DoS Attacks

As described above, current DRAM memory systems do not distinguish between the requests of different threads (i.e. cores)<sup>4</sup>. Therefore, multi-core systems are vulnerable to DoS attacks that exploit unfairness in the memory system. Requests from a thread with a particular access pattern can get prioritized by the memory access scheduler over requests from other threads, thereby causing the other threads to experience very long delays. We find that there are two major reasons why one thread can deny service to another in current DRAM memory systems:

1. **Unfairness of row-hit-first scheduling:** A thread whose accesses result in row hits gets higher priority compared to a thread whose accesses result in row conflicts. We call an access pattern that mainly results in row hits as a pattern with *high row-buffer locality*. Thus, an application that has a high row-buffer locality (e.g. one that is streaming through memory) can significantly delay another application with low row-buffer locality if they happen to be accessing the same DRAM banks.
2. **Unfairness of oldest-first scheduling:** Oldest-first scheduling implicitly gives higher priority to those threads that can generate memory requests at a faster rate than others. Such aggressive threads can flood the memory system with requests at a faster rate than the memory system can service. As such, aggressive threads can fill the memory system's buffers with their requests, while less memory-intensive threads are blocked from the memory system until all the earlier-arriving requests from the aggressive threads are serviced.

Based on this understanding, it is possible to develop a memory performance hog that effectively denies service to other threads. In the next section, we describe an example MPH and show its impact on another application.

## 3 Motivation: Examples of Denial of Memory Service in Existing Multi-Cores

In this section, we present measurements from real systems to demonstrate that Denial of Memory Service attacks are possible in existing multi-core systems.

### 3.1 Applications

We consider two applications to motivate the problem. One is a modified version of the popular *stream* benchmark [21], an application that streams through memory and performs operations on two one-dimensional arrays. The arrays in *stream* are sized such that they are much

<sup>4</sup>We assume, without loss of generality, one core can execute one thread.

```

// initialize array a, b
for j = 0; j < N; j++
    inde = j // streaming index
    ...
    for j = 0; j < N; j++
        a[inde] = j; b[inde] = j
    for j = 0; j < N; j++
        b[inde] = j; calar = a[inde]
    ...

```

(a) STREAM

```

// initialize array a, b
for j = 0; j < N; j++
    inde = j * rand // random index
    ...
    for j = 0; j < N; j++
        a[inde] = j; b[inde] = j
    for j = 0; j < N; j++
        b[inde] = j; calar = a[inde]
    ...

```

(b) RDARRAY

Figure 3: Major loops of the *stream* (a) and *rdarray* (b) programs

larger than the L2 cache on a core. Each array consists of 2.5M 128-byte elements.<sup>5</sup> *Stream* (Figure 3(a)) has very high row-buffer locality since consecutive cache misses almost always access the same row (limited only by the size of the row-buffer). Even though we cannot directly measure the row-buffer hit rate in our real experimental system (because hardware does not directly provide this information), our simulations show that 96% of all memory requests in *stream* result in row-hits.

The other application, called *rdarray*, is almost the exact opposite of *stream* in terms of its row-buffer locality. Its pseudo-code is shown in Figure 3(b). Although it performs the same operations on two very large arrays (each consisting of 2.5M 128-byte elements), *rdarray* accesses the arrays in a pseudo-random fashion. The array indices accessed in each iteration of the benchmark’s main loop are determined using a pseudo-random number generator. Consequently, this benchmark has very low row-buffer locality; the likelihood that any two outstanding L2 cache misses in the memory request buffer are to the same row in a bank is low due to the pseudo-random generation of array indices. Our simulations show that 97% of all requests in *rdarray* result in row-conflicts.

## 3.2 Measurements

We ran the two applications alone and together on two existing multi-core systems and one simulated future multi-core system.

### 3.2.1 A Dual-core System

The first system we examine is an Intel Pentium D 930 [17] based dual-core system with 2GB SDRAM. In this system each core has an L2 cache size of 2MB. Only the DRAM memory system is shared between the two cores. The operating system is Windows XP Professional.<sup>6</sup> All the experiments were performed when

<sup>5</sup>Even though the elements are 128-byte, each iteration of the main loop operates on only one 4-byte integer in the 128-byte element. We use 128-byte elements to ensure that consecutive accesses miss in the cache and exercise the DRAM memory system.

<sup>6</sup>We also repeated the same experiments in (1) the same system with the RedHat Fedora Core 6 operating system and (2) an Intel Core Duo based dual-core system running RedHat Fedora Core 6. We found the results to be almost exactly the same as those reported.

the systems were unloaded as much as possible. To account for possible variability due to system state, each run was repeated 10 times and the execution time results were averaged (error bars show the variance across the repeated runs). Each application’s main loop consists of  $N = 2.5 \cdot 10^6$  iterations and was repeated 1000 times in the measurements.

Figure 4(a) shows the normalized execution time of *stream* when run (1) alone, (2) concurrently with another copy of *stream*, and (3) concurrently with *rdarray*. Figure 4(b) shows the normalized execution time of *rdarray* when run (1) alone, (2) concurrently with another copy of *rdarray*, and (3) concurrently with *stream*.

When *stream* and *rdarray* execute concurrently on the two different cores, *stream* is slowed down by only 18%. In contrast, *rdarray* experiences a dramatic slowdown: its execution time increases by up to 190%. Hence, *stream* effectively denies memory service to *rdarray* without being significantly slowed down itself.

We hypothesize that this behavior is due to the row-hit-first scheduling policy in the DRAM memory controller. As most of *stream*’s memory requests hit in the row-buffer, they are prioritized over *rdarray*’s requests, most of which result in row conflicts. Consequently, *rdarray* is denied access to the DRAM banks that are being accessed by *stream* until the *stream* program’s access pattern moves on to another bank. With a row size of 8KB and a cache line size of 64B, 128 (=8KB/64B) of *stream*’s memory requests can be serviced by a DRAM bank before *rdarray* is allowed to access that bank!<sup>7</sup> Thus, due to the thread-unfair implementation of the DRAM memory system, *stream* can act as an MPH against *rdarray*.

Note that the slowdown *rdarray* experiences when run

<sup>7</sup>Note that we do not know the exact details of the DRAM memory controller and scheduling algorithm that is implemented in the existing systems. These details are not made public in either Intel’s or AMD’s documentation. Therefore, we hypothesize about the causes of the behavior based on public information available on DRAM memory systems - and later support our hypotheses with our simulation infrastructure (see Section 6). It could be possible that existing systems have a threshold up to which younger requests can be ordered over older requests as described in a patent [33], but even so our experiments suggest that memory performance attacks are still possible in existing multi-core systems.

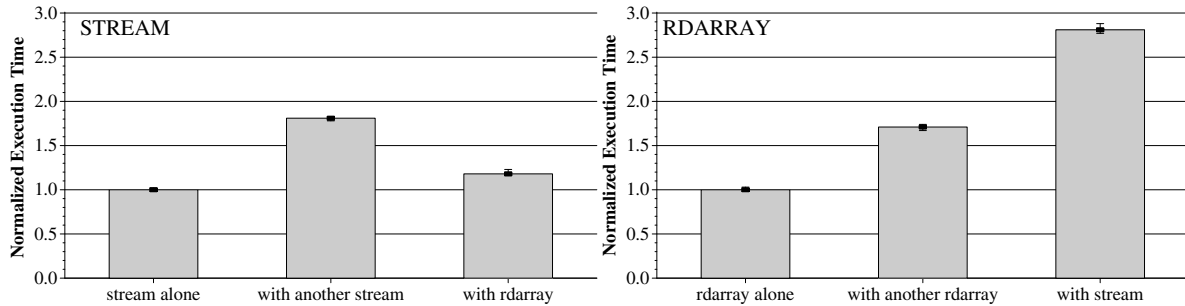


Figure 4: Normalized execution time of (a) *stream* and (b) *rdarray* when run alone/together on a dual-core system

with *stream* (2.90X) is much greater than the slowdown it experiences when run with another copy of *rdarray* (1.71X). Because neither copy of *rdarray* has good row-buffer locality, another copy of *rdarray* cannot deny service to *rdarray* by holding on to a row-buffer for a long time. In this case, the performance loss comes from increased bank conflicts and contention in the DRAM bus.

On the other hand, the slowdown *stream* experiences when run with *rdarray* is significantly smaller than the slowdown it experiences when run with another copy of *stream*. When two copies of *stream* run together they are both able to deny access to each other because they both have very high row-buffer locality. Because the rates at which both *streams* generate memory requests are the same, the slowdown is not as high as *rdarray*'s slowdown with *stream*: copies of *stream* take turns in denying access to each other (in different DRAM banks) whereas *stream* always denies access to *rdarray* (in all DRAM banks).

### 3.2.2 A Dual Dual-core System

The second system we examine is a dual dual-core AMD Opteron 275 [1] system with 4GB SDRAM. In this system, only the DRAM memory system is shared between a total of four cores. Each core has an L2 cache size of 1 MB. The operating system used was RedHat Fedora Core 5. Figure 5(a) shows the normalized execution time of *stream* when run (1) alone, (2) with one copy of *rdarray*, (3) with 2 copies of *rdarray*, (4) with 3 copies of *rdarray*, and (5) with 3 other copies of *stream*. Figure 5(b) shows the normalized execution time of *rdarray* in similar but “dual” setups.

Similar to the results shown for the dual-core Intel system, the performance of *rdarray* degrades much more significantly than the performance of *stream* when the two applications are executed together on the 4-core AMD system. In fact, *stream* slows down by only 48% when it is executed concurrently with 3 copies of *rdarray*. In contrast, *rdarray* slows down by 408% when running concurrently with 3 copies of *stream*. Again, we hypothesize that this difference in slowdowns is due to the row-hit-first policy employed in the DRAM controller.

### 3.2.3 A Simulated 16-core System

While the problem of MPHs is severe even in current dual- or dual-dual-core systems, it will be significantly aggravated in future multi-core systems consisting of many more cores. To demonstrate the severity of the problem, Figure 6 shows the normalized execution time of *stream* and *rdarray* when run concurrently with 15 copies of *stream* or 15 copies of *rdarray*, along with their normalized execution times when 8 copies of each application are run together. Note that our simulation methodology and simulator parameters are described in Section 6.1. In a 16-core system, our memory performance hog, *stream*, slows down *rdarray* by 14.6X while *rdarray* slows down *stream* by only 4.4X. Hence, *stream* is an even more effective performance hog in a 16-core system, indicating that the problem of “memory performance attacks” will become more severe in the future if the memory system is not adjusted to prevent them.

## 4 Towards a Solution: Fairness in DRAM Memory Systems

The fundamental unifying cause of the attacks demonstrated in the previous section is *unfairness* in the shared DRAM memory system. The problem is that the memory system cannot distinguish whether a harmful memory access pattern issued by a thread is due to a malicious attack, due to erroneous programming, or simply a necessary memory behavior of a specific application. Therefore, the best the DRAM memory scheduler can do is to *contain and limit* memory attacks by providing fairness among different threads.

**Difficulty of Defining DRAM Fairness:** But what exactly constitutes fairness in DRAM memory systems? As it turns out, answering this question is non-trivial and coming up with a reasonable definition is somewhat problematic. For instance, simple algorithms that schedule requests in such a way that memory latencies are equally distributed among different threads disregard the fact that different threads have different amounts of row-buffer locality. As a consequence, such *equal-latency scheduling algorithms* will unduly slow down threads

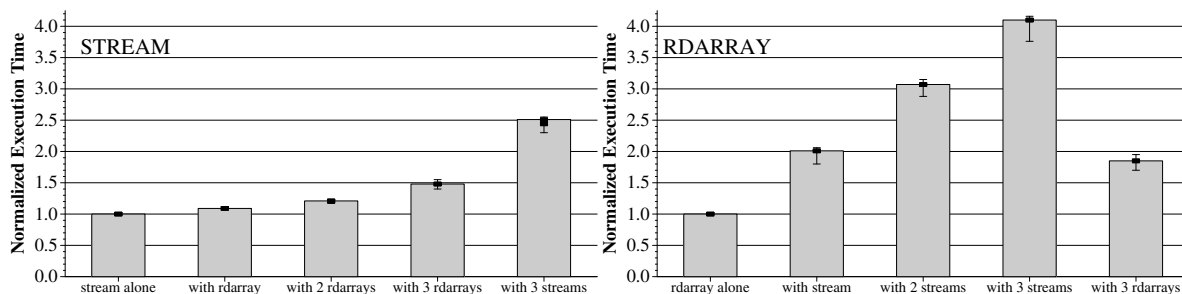


Figure 5: Slowdown of (a) *stream* and (b) *rdarray* when run alone/together on a dual dual-core system

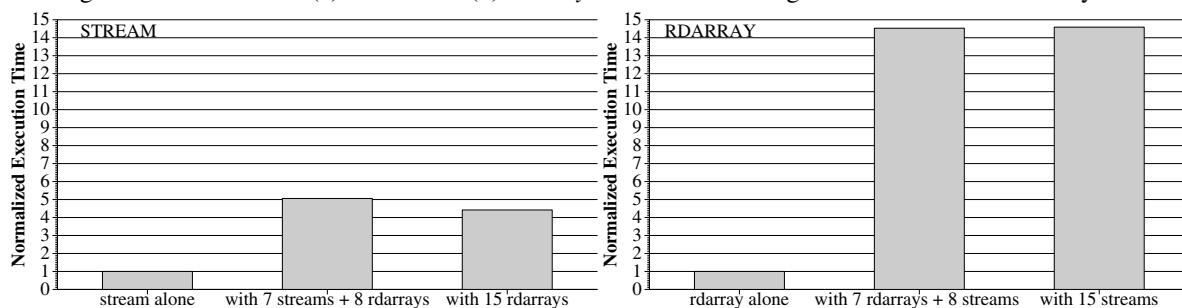


Figure 6: Slowdown of (a) *stream* and (b) *rdarray* when run alone and together on a simulated 16-core system

that have high row-buffer locality and prioritize threads that have poor row-buffer locality. Whereas the standard FR-FCFS scheduling algorithm can starve threads with poor row-buffer locality (Section 2.3), any algorithm seeking egalitarian memory fairness would unfairly punish “well-behaving” threads with good row-buffer locality. Neither of the two options therefore rules out unfairness and the possibility of memory attacks.

Another challenge is that DRAM memory systems have a notion of *state* (consisting of the currently buffered rows in each bank). For this reason, well-studied notions of fairness that deal with stateless systems cannot be applied in our setting. In *network fair queuing* [24, 40, 3], for example, the idea is that if  $N$  processes share a common channel with bandwidth  $B$ , every process should achieve exactly the same performance as if it had a single channel of bandwidth  $B/N$ . When mapping the same notion of fairness onto a DRAM memory system (as done in [23]), however, the memory scheduler would need to schedule requests in such a way as to guarantee the following: *In a multi-core system with  $N$  threads, no thread should run slower than the same thread on a single-core system with a DRAM memory system that runs at  $1/N$ th of the speed.* Unfortunately, because memory banks have state and row conflicts incur a higher latency than row hit accesses, this notion of fairness is ill-defined. Consider for instance two threads in a dual-core system that constantly access the same bank but different rows. While each of these threads by itself has perfect row-buffer locality, running them together will inevitably result in row-buffer conflicts. Hence, it is impossible to schedule these threads in such a way

that each thread runs at the same speed as if it ran by itself on a system at half the speed. On the other hand, requests from two threads that consistently access different banks could (almost) entirely be scheduled in parallel and there is no reason why the memory scheduler should be allowed to slow these threads down by a factor of 2.

In summary, in the context of memory systems, notions of fairness—such as network fair queuing—that attempt to equalize the latencies experienced by different threads are unsuitable. In a DRAM memory system, it is neither possible to achieve such a fairness nor would achieving it significantly reduce the risk of memory performance attacks. In Section 4.1, we will present a novel definition of DRAM fairness that takes into account the inherent row-buffer locality of threads and attempts to balance the “relative slowdowns”.

**The Idleness Problem:** In addition to the above observations, it is important to observe that any scheme that tries to balance latencies between threads runs into the risk of what we call the *idleness problem*. Threads that are temporarily idle (not issuing many memory requests, for instance due to a computation-intensive program phase) will be slowed down when returning to a more memory intensive access pattern. On the other hand, in certain solutions based on network fair queuing [23], a memory hog could intentionally issue no or few memory requests for a period of time. During that time, other threads could “move ahead” at a proportionally lower latency, such that, when the malicious thread returns to an intensive access pattern, it is temporarily prioritized and normal threads are blocked. The idleness problem therefore poses a severe security risk: By ex-

ploiting it, an attacking memory hog could temporarily slow down or even block time-critical applications with high performance stability requirements from memory.

## 4.1 Fair Memory Scheduling: A Model

As discussed, standard notions of fairness fail in providing fair execution and hence, security, when mapping them onto shared memory systems. The crucial insight that leads to a better notion of fairness is that we need to *dissect* the memory latency experienced by a thread into two parts: First, the latency that is inherent to the thread itself (depending on its row-buffer locality) and second, the latency that is caused by contention with other threads in the shared DRAM memory system. A fair memory system should—unlike the approaches so far—schedule requests in such a way that the *second* latency component is fairly distributed, while the first component remains untouched. With this, it is clear why our novel notion of *DRAM shared memory fairness* is based on the following intuition: *In a multi-core system with  $N$  threads, no thread should suffer more relative performance slowdown—compared to the performance it gets if it used the same memory system by itself—than any other thread.* Because each thread’s slowdown is thus measured against its own baseline performance (single execution on the same system), this notion of fairness successfully dissects the two components of latency and takes into account the inherent characteristics of each thread.

In more technical terms, we consider a measure  $\chi_i$  for each currently executed thread  $i$ .<sup>8</sup> This measure captures the price (in terms of relative additional latency) a thread  $i$  pays because the shared memory system is used by multiple threads in parallel in a multi-core architecture. In order to provide fairness and contain the risk of denial of memory service attacks, the memory controller should schedule outstanding requests in the buffer in such a way that the  $\chi_i$  values are as balanced as possible. Such a scheduling will ensure that each thread only suffers a fair amount of additional latency that is caused by the parallel usage of the shared memory system.

**Formal Definition:** Our definition of the measure  $\chi_i$  is based on the notion of *cumulated bank-latency*  $L_{i,b}$  that we define as follows.

**Definition 4.1.** *For each thread  $i$  and bank  $b$ , the cumulated bank-latency  $L_{i,b}$  is the number of memory cycles during which there exists an outstanding memory request by thread  $i$  for bank  $b$  in the memory request buffer. The cumulated latency of a thread  $L_i = \sum_b L_{i,b}$  is the sum of all cumulated bank-latencies of thread  $i$ .*

<sup>8</sup>The DRAM memory system only keeps track of threads that are currently issuing requests.

The motivation for this formulation of  $L_{i,b}$  is best seen when considering latencies on the level of individual memory requests. Consider a thread  $i$  and let  $R_{i,b}^k$  denote the  $k$ th memory request of thread  $i$  that accesses bank  $b$ . Each such request  $R_{i,b}^k$  is associated with three specific times: Its arrival time  $a_{i,b}^k$  when it is entered into the request buffer; its finish time  $f_{i,b}^k$ , when it is completely serviced by the bank and sent to processor  $i$ ’s cache; and finally, the request’s *activation time*

$$s_{i,b}^k := \max\{f_{i,b}^{k-1}, a_{i,b}^k\}.$$

This is the earliest time when request  $R_{i,b}^k$  could be scheduled by the bank scheduler. It is the larger of its arrival time and the finish time of the previous request  $R_{i,b}^{k-1}$  that was issued by the same thread to the same bank. A request’s activation time marks the point in time from which on  $R_{i,b}^k$  is responsible for the ensuing latency of thread  $i$ ; before  $s_{i,b}^k$ , the request was either not sent to the memory system or an earlier request to the same bank by the same thread was generating the latency. With these definitions, the *amortized latency*  $\ell_{i,b}^k$  of request  $R_{i,b}^k$  is the difference between its finish time and its activation time, i.e.,  $\ell_{i,b}^k = f_{i,b}^k - s_{i,b}^k$ . By the definition of the activation time  $s_{i,b}^k$ , it is clear that at any point in time, the amortized latency of exactly one outstanding request is increasing (if there is at least one in the request buffer). Hence, when describing time in terms of executed memory cycles, our definition of cumulated bank-latency  $L_{i,b}$  corresponds exactly to the sum over all amortized latencies to this bank, i.e.,  $L_{i,b} = \sum_k \ell_{i,b}^k$ .

In order to compute the experienced slowdown of each thread, we compare the actual experienced cumulated latency  $L_i$  of each thread  $i$  to an imaginary, *ideal single-core cumulated latency*  $\tilde{L}_i$  that serves as a baseline. This latency  $\tilde{L}_i$  is the minimal cumulated latency that thread  $i$  would have accrued if it had run as the only thread in the system using the same DRAM memory; it captures the latency component of  $L_i$  that is inherent to the thread itself and not caused by contention with other threads. Hence, threads with good and bad row-buffer locality have small and large  $\tilde{L}_i$ , respectively. The measure  $\chi_i$  that captures the relative slowdown of thread  $i$  caused by multi-core parallelism can now be defined as follows.

**Definition 4.2.** *For a thread  $i$ , the DRAM memory slowdown index  $\chi_i$  is the ratio between its cumulated latency  $L_i$  and its ideal single-core cumulated latency  $\tilde{L}_i$ .<sup>9</sup>*

<sup>9</sup>Notice that our definitions do not take into account the service and waiting times of the shared DRAM bus and across-bank scheduling. Both our definition of fairness as well as our algorithm presented in Section 5 can be extended to take into account these and other more subtle hardware issues. As the main goal of this paper point out and investigate potential security risks caused by DRAM unfairness, our model abstracts away numerous aspects of secondary importance because our definition provides a good approximation.

$$\chi_i := L_i / \tilde{L}_i.$$

Finally, we define the **DRAM unfairness**  $\Psi$  of a DRAM memory system as the ratio between the maximum and minimum slowdown index over all currently executed threads in the system:

$$\Psi := \frac{\max_i \chi_i}{\min_j \chi_j}$$

The “ideal” DRAM unfairness index  $\Psi = 1$  is achieved if all threads experience exactly the same slowdown; the higher  $\Psi$ , the more unbalanced is the experienced slowdown of different threads. The goal of a fair memory access scheduling algorithm is therefore to achieve a  $\Psi$  that is as close to 1 as possible. This ensures that no thread is over-proportionally slowed down due to the shared nature of DRAM memory in multi-core systems.

Notice that by taking into account the different row-buffer localities of different threads, our definition of DRAM unfairness prevents punishing threads for having either good or bad memory access behavior. Hence, a scheduling algorithm that achieves low DRAM unfairness mitigates the risk that any thread in the system, regardless of its bank and row access pattern, is unduly bogged down by other threads. Notice further that DRAM unfairness is virtually unaffected by the idleness problem, because both cumulated latencies  $L_i$  and ideal single-core cumulated latencies  $\tilde{L}_i$  are only accrued when there are requests in the memory request buffer.

**Short-Term vs. Long-Term Fairness:** So far, the aspect of time-scale has remained unspecified in our definition of DRAM-unfairness. Both  $L_i$  and  $\tilde{L}_i$  continue to increase throughout the lifetime of a thread. Consequently, a short-term unfair treatment of a thread would have increasingly little impact on its slowdown index  $\chi_i$ . While still providing long-term fairness, threads that have been running for a long time could become vulnerable to short-term DoS attacks even if the scheduling algorithm enforced an upper bound on DRAM unfairness  $\Psi$ . In this way, delay-sensitive applications could be blocked from DRAM memory for limited periods of time.

We therefore generalize all our definitions to include an additional parameter  $T$  that denotes the time-scale for which the definitions apply. In particular,  $L_i(T)$  and  $\tilde{L}_i(T)$  are the maximum (ideal single-core) cumulated latencies over all time-intervals of duration  $T$  during which thread  $i$  is active. Similarly,  $\chi_i(T)$  and  $\Psi(T)$  are defined as the maximum values over all time-intervals of length  $T$ . The parameter  $T$  in these definitions determines how short- or long-term the considered fairness is. In particular, a memory scheduling algorithm with good long term fairness will have small  $\Psi(T)$  for large  $T$ , but possibly large  $\Psi(T')$  for smaller  $T'$ . In view of the security issues raised in this paper, it is clear that a memory scheduling algorithm should aim at achieving small  $\Psi(T)$  for both small and large  $T$ .

## 5 Our Solution

In this section, we propose FairMem, a new fair memory scheduling algorithm that achieves good fairness according to the definition in Section 4 and hence, reduces the risk of memory-related DoS attacks.

### 5.1 Basic Idea

The reason why MPHs can exist in multi-core systems is the unfairness in current memory access schedulers. Therefore, the idea of our new scheduling algorithm is to enforce fairness by balancing the relative memory-related slowdowns experienced by different threads. The algorithm schedules requests in such a way that each thread experiences a similar degree of memory-related slowdown relative to its performance when run alone.

In order to achieve this goal, the algorithm maintains a value ( $\chi_i$  in our model of Section 4.1) that characterizes the relative slowdown of each thread. As long as all threads have roughly the same slowdown, the algorithm schedules requests using the regular FR-FCFS mechanism. When the slowdowns of different threads start diverging and the difference exceeds a certain threshold (i.e., when  $\Psi$  becomes too large), however, the algorithm switches to an alternative scheduling mechanism and starts prioritizing requests issued by threads experiencing large slowdowns.

### 5.2 Fair Memory Scheduling Algorithm (FairMem)

The memory scheduling algorithm we propose for use in DRAM controllers for multi-core systems is defined by means of two input parameters,  $\alpha$  and  $\beta$ . These parameters can be used to fine-tune the involved trade-offs between fairness and throughput on the one hand ( $\alpha$ ) and short-term versus long-term fairness on the other ( $\beta$ ). More concretely,  $\alpha$  is a parameter that expresses to what extent the scheduler is allowed to optimize for DRAM throughput at the cost of fairness, i.e., how much DRAM unfairness is tolerable. The parameter  $\beta$  corresponds to the time-interval  $T$  that denotes the time-scale of the above fairness condition. In particular, the memory controller divides time into windows of duration  $\beta$  and, for each thread maintains an accurate account of its accumulated latencies  $L_i(\beta)$  and  $\tilde{L}_i(\beta)$  in the current time window.<sup>10</sup>

<sup>10</sup>Notice that in principle, there are various possibilities of interpreting the term “current time window.” The simplest way is to completely reset  $L_i(\beta)$  and  $\tilde{L}_i(\beta)$  after each completion of a window. More sophisticated techniques could include maintaining multiple, say  $k$ , such windows of size  $\beta$  in parallel, each shifted in time by  $\beta/k$  memory cycles. In this case, all windows are constantly updated, but only the oldest is used for the purpose of decision-making. This could help in reducing volatility.

Instead of using the (FR-FCFS) algorithm described in Section 2.2.3, our algorithm first determines two *candidate requests* from each bank  $b$ , one according to each of the following rules:

- **Highest FR-FCFS priority:** Let  $R_{\text{FR-FCFS}}$  be the request to bank  $b$  that has the highest priority according to the FR-FCFS scheduling policy of Section 2.2.3. That is, row hits have higher priority than row conflicts, and—given this partial ordering—the oldest request is served first.
- **Highest fairness-index:** Let  $i'$  be the thread with highest current DRAM memory slowdown index  $\chi_{i'}(\beta)$  that has at least one outstanding request in the memory request buffer to bank  $b$ . Among all requests to  $b$  issued by  $i'$ , let  $R_{\text{Fair}}$  be the one with highest FR-FCFS priority.

Between these two candidates, the algorithm chooses the request to be scheduled based on the following rule:

- **Fairness-oriented Selection:** Let  $\chi_\ell(\beta)$  and  $\chi_s(\beta)$  denote largest and smallest DRAM memory slowdown index of any request in the memory request buffer for a current time window of duration  $\beta$ . If it holds that

$$\frac{\chi_\ell(\beta)}{\chi_s(\beta)} \geq \alpha$$

then  $R_{\text{Fair}}$  is selected by bank  $b$ 's scheduler and  $R_{\text{FR-FCFS}}$  otherwise.

Instead of using the oldest-across-banks-first strategy as used in current DRAM memory schedulers, selection from requests chosen by the bank schedulers is handled as follows:

**Highest-DRAM-fairness-index-first across banks:**

The request with highest slowdown index  $\chi_i(\beta)$  among all selected bank-requests is sent on the shared DRAM bus.

In principle, the algorithm is built to ensure that at no time DRAM unfairness  $\Psi(\beta)$  exceeds the parameter  $\alpha$ . Whenever there is the risk of exceeding this threshold, the memory controller will switch to a mode in which it starts prioritizing threads with higher  $\chi_i$  values, which decreases  $\chi_i$ . It also increases the  $\chi_j$  values of threads that have had little slowdown so far. Consequently, this strategy balances large and small slowdowns, which decreases DRAM unfairness and—as shown in Section 6—keeps potential memory-related DoS attacks in check.

Notice that this algorithm does not—in fact, cannot—guarantee that the DRAM unfairness  $\Psi$  does stay below the predetermined threshold  $\alpha$  at all times. The impossibility of this can be seen when considering the corner-case  $\alpha = 1$ . In this case, a violation occurs after the first request regardless of which request is scheduled by the algorithm. On the other hand, the algorithm always attempts to keep the necessary violations to a minimum.

Another advantage of our scheme is that an approximate version of it lends itself to efficient implementation in hardware. Finally, notice that our algorithm is robust with regard to the *idleness problem* mentioned in Section 4. In particular, neither  $L_i$  nor  $\tilde{L}_i$  is increased or decreased if a thread has no outstanding memory requests in the request buffer. Hence, not issuing any requests for some period of time (either intentionally or unintentionally) does not affect this or any other thread's priority in the buffer.

### 5.3 Hardware Implementations

The algorithm as described so far is abstract in the sense that it assumes a memory controller that always has full knowledge of every active (currently-executed) thread's  $L_i$  and  $\tilde{L}_i$ . In this section, we show how this exact scheme could be implemented, and we also briefly discuss a more efficient practical hardware implementation.

**Exact Implementation:** Theoretically, it is possible to ensure that the memory controller always keeps accurate information of  $L_i(\beta)$  and  $\tilde{L}_i(\beta)$ . Keeping track of  $L_i(\beta)$  for each thread is simple. For each active thread, a counter maintains the number of memory cycles during which at least one request of this thread is buffered for each bank. After completion of the window  $\beta$  (or when a new thread is scheduled on a core), counters are reset. The more difficult part of maintaining an accurate account of  $\tilde{L}_i(\beta)$  can be done as follows: At all times, maintain for each active thread  $i$  and for each bank the row that would currently be in the row-buffer if  $i$  had been the only thread using the DRAM memory system. This can be done by simulating an FR-FCFS priority scheme for each thread and bank that ignores all requests issued by threads other than  $i$ . The  $\tilde{\ell}_{i,b}^k$  latency of each request  $R_{i,b}^k$  then corresponds to the latency this request would have caused if DRAM memory was not shared. Whenever a request is served, the memory controller can add this “ideal latency” to the corresponding  $\tilde{L}_{i,b}(\beta)$  of that thread and—if necessary—update the simulated state of the row-buffer accordingly. For instance, assume that a request  $R_{i,b}^k$  is served, but results in a row conflict. Assume further that the same request would have been a row hit, if thread  $i$  had run by itself, i.e.,  $R_{i,b}^{k-1}$  accesses the same row as  $R_{i,b}^k$ . In this case,  $\tilde{L}_{i,b}(\beta)$  is increased by row-hit latency  $T_{\text{hit}}$ , whereas  $L_{i,b}(\beta)$  is increased by the bank-conflict latency  $T_{\text{conf}}$ . By thus “simulating” its own execution for each thread, the memory controller obtains accurate information for all  $\tilde{L}_{i,b}(\beta)$ .

The obvious problem with the above implementation is that it is expensive in terms of hardware overhead. It requires maintaining at least one counter for each core  $\times$  bank pair. Similarly severe, it requires one divider per core in order to compute the value  $\chi_i(\beta) =$

$L_i(\beta)/\tilde{L}_i(\beta)$  for the thread that is currently running on that core in every memory cycle. Fortunately, much less expensive hardware implementations are possible because the memory controller does not need to know the exact values of  $L_{i,b}$  and  $\tilde{L}_{i,b}$  at any given moment. Instead, using reasonably accurate approximate values suffices to maintain an excellent level of fairness and security.

**Reduce counters by sampling:** Using sampling techniques, the number of counters that need to be maintained can be reduced from  $O(\#Banks \times \#Cores)$  to  $O(\#Cores)$  with only little loss in accuracy. Briefly, the idea is the following. For each core and its active thread, we keep two counters  $S_i$  and  $H_i$  denoting the number of samples and sampled hits, respectively. Instead of keeping track of the exact row that would be open in the row-buffer if a thread  $i$  was running alone, we randomly sample a subset of requests  $R_{i,b}^k$  issued by thread  $i$  and check whether the next request by  $i$  to the same bank,  $R_{i,b}^{k+1}$ , is for the same row. If so, the memory controller increases both  $S_i$  and  $H_i$ , otherwise, only  $S_i$  is increased. Requests  $R_{i,b'}^q$  to different banks  $b' \neq b$  served between  $R_{i,b}^k$  and  $R_{i,b}^{k+1}$  are ignored. Finally, if none of the  $Q$  requests of thread  $i$  following  $R_{i,b}^k$  go to bank  $b$ , the sample is discarded, neither  $S_i$  nor  $H_i$  is increased, and a new sample request is taken. With this technique, the probability  $H_i/S_i$  that a request results in a row hit gives the memory controller a reasonably accurate picture of each thread's row-buffer locality. An approximation of  $\tilde{L}_i$  can thus be maintained by adding the expected amortized latency to it whenever a request is served, i.e.,

$$\tilde{L}_i^{new} := \tilde{L}_i^{old} + (H_i/S_i \cdot T_{hit} + (1 - H_i/S_i) \cdot T_{conf}).$$

**Reuse dividers:** The ideal scheme employs  $O(\#Cores)$  hardware dividers, which significantly increases the memory controller's energy consumption. Instead, a single divider can be used for all cores by assigning individual threads to it in a round robin fashion. That is, while the slowdowns  $L_i(\beta)$  and  $\tilde{L}_i(\beta)$  can be updated in every memory cycle, their quotient  $\chi_i(\beta)$  is recomputed in intervals.

## 6 Evaluation

### 6.1 Experimental Methodology

We evaluate our solution using a detailed processor and memory system simulator based on the Pin dynamic binary instrumentation tool [20]. Our in-house instruction-level performance simulator can simulate applications compiled for the x86 instruction set architecture. We simulate the memory system in detail using a model loosely based on DRAMsim [36]. Both our processor model and the memory model mimic the design of a modern high-performance dual-core proces-

sor loosely based on the Intel Pentium M [11]. The size/bandwidth/latency/capacity of different processor structures along with the number of cores and other structures are parameters to the simulator. The simulator faithfully models the bandwidth, latency, and capacity of each buffer, bus, and structure in the memory subsystem (including the caches, memory controller, DRAM buses, and DRAM banks). The relevant parameters of the modeled baseline processor are shown in Table 1. Unless otherwise stated, all evaluations in this section are performed on a simulated dual-core system using these parameters. For our measurements with the FairMem system presented in Section 5, the parameters are set to  $\alpha = 1.025$  and  $\beta = 10^5$ .

We simulate each application for 100 million x86 instructions. The portions of applications that are simulated are determined using the SimPoint tool [32], which selects simulation points in the application that are representative of the application's behavior as a whole. Our applications include *stream* and *rdarray* (described in Section 3), several large benchmarks from the SPEC CPU2000 benchmark suite [34], and one memory-intensive benchmark from the Olden suite [31]. These applications are described in Table 2.

## 6.2 Evaluation Results

### 6.2.1 Dual-core Systems

**Two microbenchmark applications - *stream* and *rdarray*:** Figure 7 shows the normalized execution time of *stream* and *rdarray* applications when run alone or together using either the baseline FR-FCFS or our FairMem memory scheduling algorithms. Execution time of each application is normalized to the execution time they experience when they are run alone using the FR-FCFS scheduling algorithm (This is true for all normalized results in this paper). When *stream* and *rdarray* are run together on the baseline system, *stream*—which acts as an MPH—experiences a slowdown of only 1.22X whereas *rdarray* slows down by 2.45X. In contrast, a memory controller that uses our FairMem algorithm prevents *stream* from behaving like an MPH against *rdarray*—both applications experience similar slowdowns when run together. FairMem does not significantly affect performance when the applications are run alone or when run with identical copies of themselves (i.e. when memory performance is not unfairly impacted). These experiments show that our simulated system closely matches the behavior we observe in an existing dual-core system (Figure 4), and that FairMem successfully provides fairness among threads. Next, we show that with real applications, the effect of an MPH can be drastic.

**Effect on real applications:** Figure 8 shows the normalized execution time of 8 different pairs of applications when run alone or together using either the baseline FR-

Processor pipeline	4 GHz processor, 128-entry instruction window, 12-stage pipeline
Fetch/Execute width per core	3 instructions can be fetched/executed every cycle; only 1 can be a memory operation
L1 Caches	32 K-byte per-core, 4-way set associative, 32-byte block size, 2-cycle latency
L2 Caches	512 K-byte per core, 8-way set associative, 32-byte block size, 12-cycle latency
Memory controller	128 request buffer entries, FR-FCFS baseline scheduling policy, runs at 2 GHz
DRAM parameters	8 banks, 2K-byte row-buffer
DRAM latency (round-trip L2 miss latency)	row-buffer hit: 50ns (200 cycles), closed: 75ns (300 cycles), conflict: 100ns (400 cycles)

Table 1: Baseline processor configuration

Benchmark	Suite	Brief description	Base performance	L2-misses per 1K inst.	row-buffer hit rate
stream	Microbenchmark	Streaming on 32-byte-element arrays	46.30 cycles/inst.	629.65	96%
rdarray	Microbenchmark	Random access on arrays	56.29 cycles/inst.	629.18	3%
small-stream	Microbenchmark	Streaming on 4-byte-element arrays	13.86 cycles/inst.	71.43	97%
art	SPEC 2000 FP	Object recognition in thermal image	7.85 cycles/inst.	70.82	88%
crafty	SPEC 2000 INT	Chess game	0.64 cycles/inst.	0.35	15%
health	Olden	Columbian health care system simulator	7.24 cycles/inst.	83.45	27%
mcf	SPEC 2000 INT	Single-depot vehicle scheduling	4.73 cycles/inst.	45.95	51%
vpr	SPEC 2000 INT	FPGA circuit placement and routing	1.71 cycles/inst.	5.08	14%

Table 2: Evaluated applications and their performance characteristics on the baseline processor

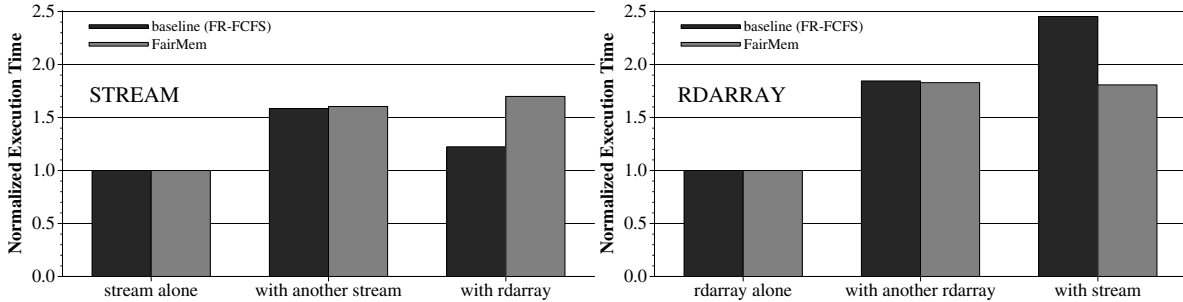


Figure 7: Slowdown of (a) *stream* and (b) *rdarray* benchmarks using FR-FCFS and our FairMem algorithm

FCFS or FairMem. The results show that 1) an MPH can severely damage the performance of another application, and 2) our FairMem algorithm is effective at preventing it. For example, when *stream* and *health* are run together in the baseline system, *stream* acts as an MPH slowing down *health* by 8.6X while itself being slowed down by only 1.05X. This is because it has 7 times higher L2 miss rate and much higher row-buffer locality (96% vs. 27%) — therefore, it exploits unfairness in both row-buffer-hit first and oldest-first scheduling policies by flooding the memory system with its requests. When the two applications are run on our FairMem system, *health*’s slowdown is reduced from 8.63X to 2.28X. The figure also shows that even regular applications with high row-buffer locality can act as MPHs. For instance when *art* and *vpr* are run together in the baseline system, *art* acts as an MPH slowing down *vpr* by 2.35X while itself being slowed down by only 1.05X. When the two are run on our FairMem system, each slows down by only 1.35X; thus, *art* is no longer a performance hog.

**Effect on Throughput and Unfairness:** Table 3 shows the overall throughput (in terms of executed instructions per 1000 cycles) and DRAM unfairness (relative difference between the maximum and minimum memory-related slowdowns, defined as  $\Psi$  in Section 4) when different application combinations are executed together. In

all cases, FairMem reduces the unfairness to below 1.20 (Remember that 1.00 is the best possible  $\Psi$  value). Interestingly, in most cases, FairMem also improves overall throughput significantly. This is especially true when a very memory-intensive application (e.g. *stream*) is run with a much less memory-intensive application (e.g. *vpr*).

Providing fairness leads to higher overall system throughput because it enables better utilization of the cores (i.e. better utilization of the multi-core system). The baseline FR-FCFS algorithm significantly hinders the progress of a less memory-intensive application, whereas FairMem allows this application to stall less due to the memory system, thereby enabling it to make fast progress through its instruction stream. Hence, rather than wasting execution cycles due to unfairly-induced memory stalls, some cores are better utilized with FairMem.<sup>11</sup> On the other hand, FairMem reduces the overall throughput by 9% when two extremely memory-intensive applications, *stream* and *rdarray*, are run concurrently. In this case, enforcing fairness reduces *stream*’s data throughput without significantly increasing *rdarray*’s throughput because *rdarray* encounters L2 cache misses as frequently as *stream* (see Table 2).

<sup>11</sup> Note that the data throughput obtained from the DRAM itself may be, and usually is reduced using FairMem. However, overall throughput in terms of instructions executed per cycle usually increases.

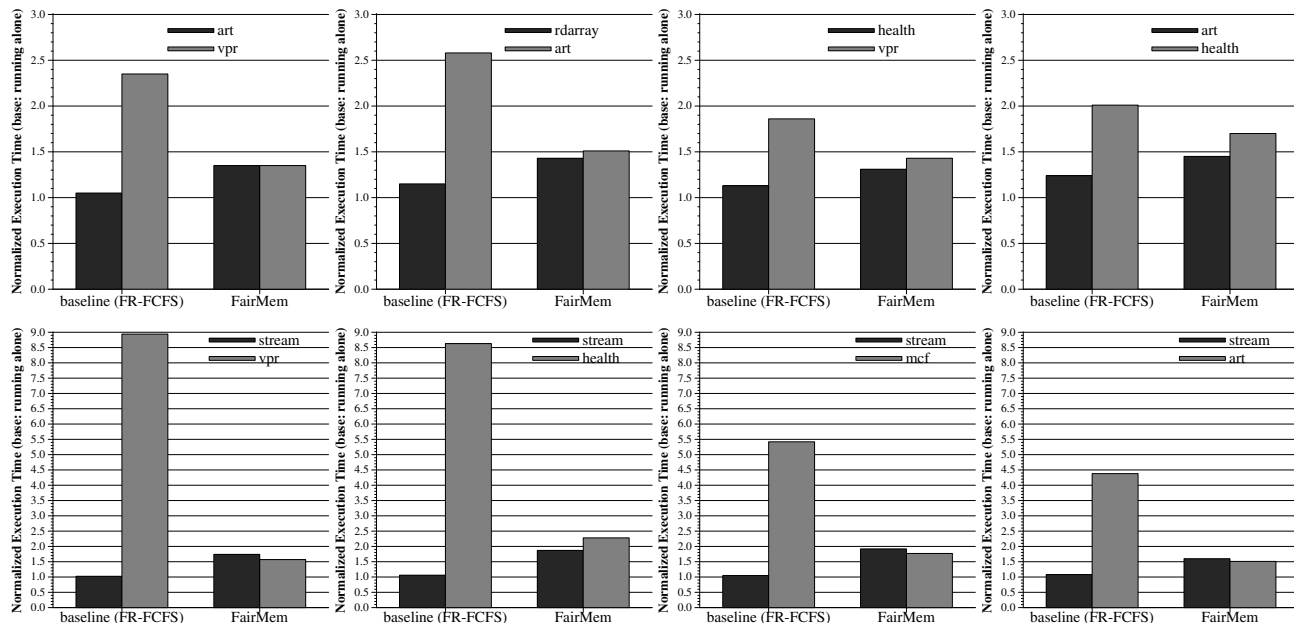


Figure 8: Slowdown of different application combinations using FR-FCFS and our FairMem algorithm

Combination	Baseline (FR-FCFS)		FairMem		Throughput improvement	Fairness improvement
	Throughput	Unfairness	Throughput	Unfairness		
stream-rdarray	24.8	2.00	22.5	1.06	0.91X	1.89X
art-vpr	401.4	2.23	513.0	1.00	1.28X	2.23X
health-vpr	463.8	1.56	508.4	1.09	1.10X	1.43X
art-health	179.3	1.62	178.5	1.15	0.99X	1.41X
rdarray-art	65.9	2.24	97.1	1.06	1.47X	2.11X
stream-health	38.0	8.14	72.5	1.18	1.91X	6.90X
stream-vpr	87.2	8.73	390.6	1.11	4.48X	7.86X
stream-mcf	63.1	5.17	117.1	1.08	1.86X	4.79X
stream-art	51.2	4.06	98.6	1.06	1.93X	3.83X

Table 3: Effect of FairMem on overall throughput (in terms of instructions per 1000 cycles) and unfairness

### 6.2.2 Effect of Row-buffer Size

From the above discussions, it is clear that the exploitation of row-buffer locality by the DRAM memory controller makes the multi-core memory system vulnerable to DoS attacks. The extent to which this vulnerability can be exploited is determined by the size of the row-buffer. In this section, we examine the impact of row-buffer size on the effectiveness of our algorithm. For these sensitivity experiments we use two real applications, *art* and *vpr*, where *art* behaves as an MPH against *vpr*.

Figure 9 shows the mutual impact of *art* and *vpr* on machines with different row-buffer sizes. Additional statistics are presented in Table 4. As row-buffer size increases, the extent to which *art* becomes a memory performance hog for *vpr* increases when FR-FCFS scheduling algorithm is used. In a system with very small, 512-byte row-buffers, *vpr* experiences a slowdown of 1.65X (versus *art*'s 1.05X). In a system with very large, 64 KB row-buffers, *vpr* experiences a slowdown of 5.50X (versus *art*'s 1.03X). Because *art* has very high row-buffer locality, a large buffer size allows its accesses to occupy a bank much longer than a small buffer size does. Hence,

*art*'s ability to deny bank service to *vpr* increases with row-buffer size. FairMem effectively contains this denial of service and results in similar slowdowns for both *art* and *vpr* (1.32X to 1.41X). It is commonly assumed that row-buffer sizes will increase in the future to allow better throughput for streaming applications [41]. As our results show, this implies that memory-related DoS attacks will become a larger problem and algorithms to prevent them will become more important.<sup>12</sup>

### 6.2.3 Effect of Number of Banks

The number of DRAM banks is another important parameter that affects how much two threads can interfere with each others' memory accesses. Figure 10 shows the impact of *art* and *vpr* on each other on machines with different number of DRAM banks. As the number of banks increases, the available parallelism in the

<sup>12</sup>Note that reducing the row-buffer size may at first seem like one way of reducing the impact of memory-related DoS attacks. However, this solution is not desirable because reducing the row-buffer size significantly reduces the memory bandwidth (hence performance) for applications with good row-buffer locality even when they are running alone or when they are not interfering with other applications.

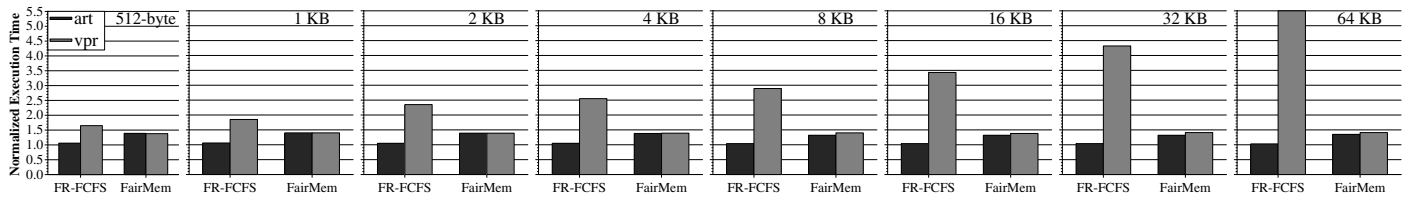


Figure 9: Normalized execution time of *art* and *vpr* when run together on processors with different row-buffer sizes. Execution time is independently normalized to each machine with different row-buffer size.

	512 B	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB
<i>art</i> 's row-buffer hit rate	56%	67%	87%	91%	92%	93%	95%	98%
<i>vpr</i> 's row-buffer hit rate	13%	15%	17%	19%	23%	28%	38%	41%
FairMem throughput improvement	1.08X	1.16X	1.28X	1.44X	1.62X	1.88X	2.23X	2.64X
FairMem fairness improvement	1.55X	1.75X	2.23X	2.42X	2.62X	3.14X	3.88X	5.13X

Table 4: Statistics for *art* and *vpr* with different row-buffer sizes

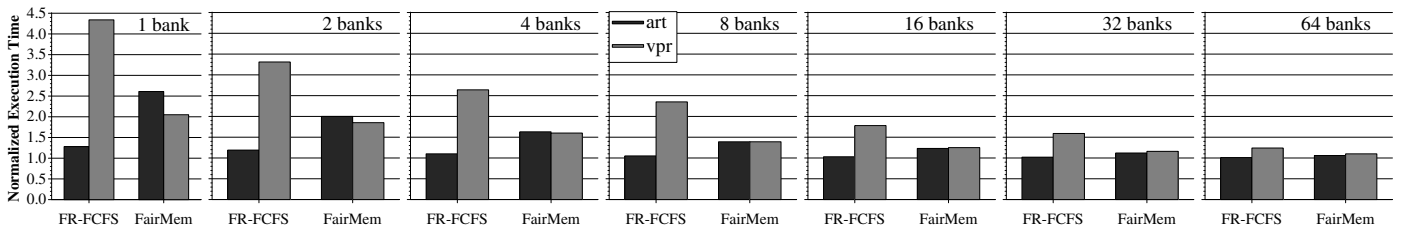


Figure 10: Slowdown of *art* and *vpr* when run together on processors with various number of DRAM banks. Execution time is independently normalized to each machine with different number of banks.

	1 bank	2 banks	4 banks	8 banks	16 banks	32 banks	64 banks
<i>art-vpr</i> base throughput (IPTC)	122	210	304	401	507	617	707
<i>art-vpr</i> FairMem throughput (IPTC)	190	287	402	513	606	690	751
FairMem throughput improvement	1.56X	1.37X	1.32X	1.28X	1.20X	1.12X	1.06X
FairMem fairness improvement	2.67X	2.57X	2.35X	2.23X	1.70X	1.50X	1.18X

Table 5: Statistics for *art-vpr* with different number of DRAM banks (IPTC: Instructions/1000-cycles)

memory system increases, and thus *art* becomes less of a performance hog; its memory requests conflict less with *vpr*'s requests. Regardless of the number of banks, our mechanism significantly mitigates the performance impact of *art* on *vpr* while at the same time improving overall throughput as shown in Table 5. Current DRAMs usually employ 4-16 banks because a larger number of banks increases the cost of the DRAM system. In a system with 4 banks, *art* slows down *vpr* by 2.64X (while itself being slowed down by only 1.10X). FairMem is able to reduce *vpr*'s slowdown to only 1.62X and improve overall throughput by 32%. In fact, Table 5 shows that FairMem achieves the same throughput on only 4 banks as the baseline scheduling algorithm on 8 banks.

#### 6.2.4 Effect of Memory Latency

Clearly, memory latency also has an impact on the vulnerability in the DRAM system. Figure 11 shows how different DRAM latencies influence the mutual performance impact of *art* and *vpr*. We vary the round-trip latency of a request that hits in the row-buffer from 50 to 1000 processor clock cycles, and scale closed/conflict latencies proportionally. As memory latency increases, the impact of *art* on *vpr* also increases. *Vpr*'s slowdown

is 1.89X with a 50-cycle latency versus 2.57X with a 1000-cycle latency. Again, FairMem reduces *art*'s impact on *vpr* for all examined memory latencies while also improving overall system throughput (Table 6). As main DRAM latencies are expected to increase in modern processors (in terms of processor clock cycles) [39], scheduling algorithms that mitigate the impact of MPHs will become more important and effective in the future.

#### 6.2.5 Effect of Number of Cores

Finally, this section analyzes FairMem within the context of 4-core and 8-core systems. Our results show that FairMem effectively mitigates the impact of MPHs while improving overall system throughput in both 4-core and 8-core systems running different application mixes with varying memory-intensiveness.

Figure 12 shows the effect of FairMem on three different application mixes run on a 4-core system. In all the mixes, *stream* and *small-stream* act as severe MPHs when run on the baseline FR-FCFS system, slowing down other applications by up to 10.4X (and at least 3.5X) while themselves being slowed down by no more than 1.10X. FairMem reduces the maximum slowdown caused by these two hogs to at most 2.98X while also

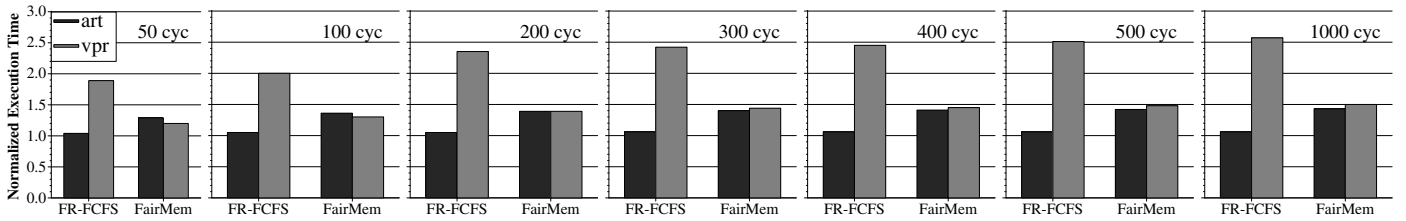


Figure 11: Slowdown of *art* and *vpr* when run together on processors with different DRAM access latencies. *Execution time is independently normalized to each machine with different number of banks. Row-buffer hit latency is denoted.*

	50 cycles	100 cycles	200 cycles	300 cycles	400 cycles	500 cycles	1000 cycles
<i>art-vpr</i> base throughput (IPTC)	1229	728	401	278	212	172	88
<i>art-vpr</i> FairMem throughput (IPTC)	1459	905	513	359	276	224	114
FairMem throughput improvement	1.19X	1.24X	1.28X	1.29X	1.30X	1.30X	1.30X
FairMem fairness improvement	1.69X	1.82X	2.23X	2.21X	2.25X	2.23X	2.22X

Table 6: Statistics for *art-vpr* with different DRAM latencies (IPTC: Instructions/1000-cycles)

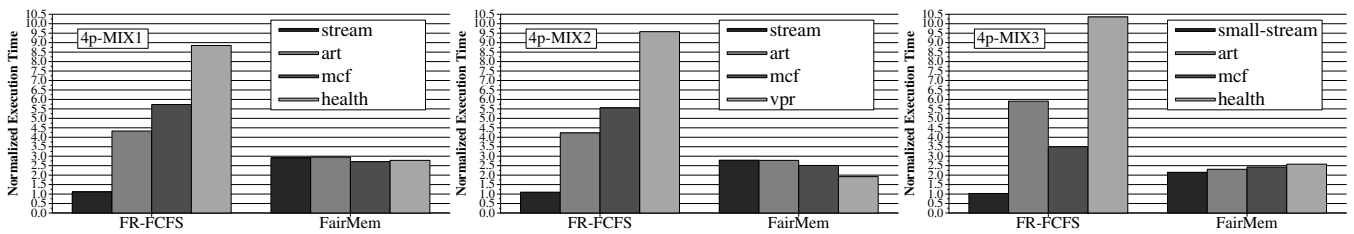


Figure 12: Effect of FR-FCFS and FairMem scheduling on different application mixes in a 4-core system

improving the overall throughput of the system (Table 7).

Figure 13 shows the effect of FairMem on three different application mixes run on an 8-core system. Again, in the baseline system, *stream* and *small-stream* act as MPHs, sometimes degrading the performance of another application by as much as 17.6X. FairMem effectively contains the negative performance impact caused by the MPHs for all three application mixes. Furthermore, it is important to observe that FairMem is also effective at isolating non-memory-intensive applications (such as *crafty* in MIX2 and MIX3) from the performance degradation caused by the MPHs. Even though *crafty* rarely generates a memory request (0.35 times per 1000 instructions), it is slowed down by 7.85X by the baseline system when run within MIX2! With FairMem *crafty*’s rare memory requests are not unfairly delayed due to a memory performance hog — and its slowdown is reduced to only 2.28X. The same effect is also observed for *crafty* in MIX3.<sup>13</sup> We conclude that FairMem provides fairness in the memory system, which improves the performance of both memory-intensive and non-memory-intensive applications that are unfairly delayed by an MPH.

## 7 Related Work

The possibility of exploiting vulnerabilities in the *software system* to deny memory allocation to other applications has been considered in a number of works. For

example, [37] describes an attack in which one process continuously allocates virtual memory and causes other processes on the same machine to run out of memory space because swap space on disk is exhausted. The “memory performance attack” we present in this paper is conceptually very different from such “memory allocation attacks” because (1) it exploits vulnerabilities in the *hardware system*, (2) it is not amenable to software solutions — the hardware algorithms must be modified to mitigate the impact of attacks, and (3) it can be caused even unintentionally by well-written, non-malicious but memory-intensive applications.

There are only few research papers that consider *hardware* security issues in computer architecture. Woo and Lee [38] describe similar shared-resource attacks that were developed concurrently with this work, but they do not show that the attacks are effective in real multi-core systems. In their work, a malicious thread tries to displace the data of another thread from the shared caches or to saturate the on-chip or off-chip bandwidth. In contrast, our attack exploits the unfairness in the DRAM memory scheduling algorithms; hence their attacks and ours are complementary.

Grunwald and Ghiasi [12] investigate the possibility of microarchitectural denial of service attacks in SMT (simultaneous multithreading) processors. They show that SMT processors exhibit a number of vulnerabilities that could be exploited by malicious threads. More specifically, they study a number of DoS attacks that affect caching behavior, including one that uses self-modifying

<sup>13</sup>Notice that 8p-MIX2 and 8p-MIX3 are much less memory intensive than 8p-MIX1. Due to this, their baseline overall throughput is significantly higher than 8p-MIX1 as shown in Table 7.

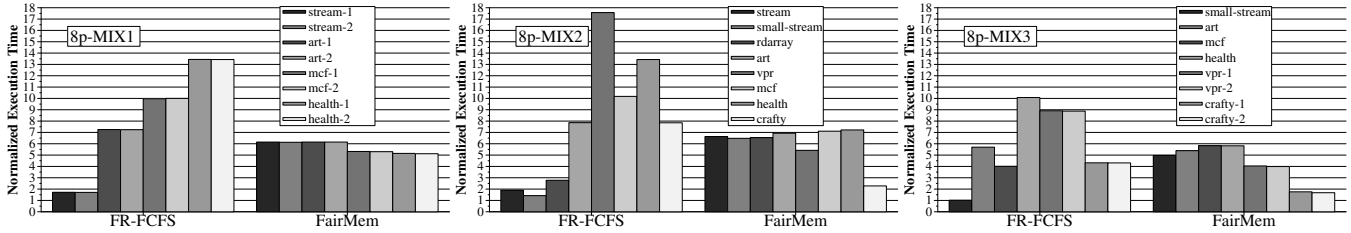


Figure 13: Effect of FR-FCFS and FairMem scheduling on different application mixes in an 8-core system

	4p-MIX1	4p-MIX2	4p-MIX3	8p-MIX1	8p-MIX2	8p-MIX3
base throughput (IPTC)	107	156	163	131	625	1793
FairMem throughput (IPTC)	179	338	234	189	1233	2809
base unfairness ( $\Psi$ )	8.05	8.71	10.98	7.89	13.56	10.11
FairMem unfairness ( $\Psi$ )	1.09	1.32	1.21	1.18	1.34	1.32
FairMem throughput improvement	1.67X	2.17X	1.44X	1.44X	1.97X	1.57X
FairMem fairness improvement	7.39X	6.60X	9.07X	6.69X	10.11X	7.66X

Table 7: Throughput and fairness statistics for 4-core and 8-core systems

code to cause the trace cache to be flushed. The authors then propose counter-measures that ensure fair pipeline utilization. The work of Hasan et al. [13] studies in a simulator the possibility of so-called *heat stroke* attacks that repeatedly access a shared resource to create a hot spot at the resource, thus slowing down the SMT pipeline. The authors propose a solution that selectively slows down malicious threads. These two papers present involved ways of “hacking” existing systems using sophisticated techniques such as self-modifying code or identifying on-chip hardware resources that can heat up. In contrast, our paper describes a more prevalent problem: a trivial type of attack that could be easily developed by anyone who writes a program. In fact, even existing simple applications may behave like memory performance hogs and future multi-core systems are bound to become even more vulnerable to MPHs. In addition, neither of the above works consider vulnerabilities in shared DRAM memory in multi-core architectures.

The FR-FCFS scheduling algorithm implemented in many current single-core and multi-core systems was studied in [30, 29, 15, 23], and its best implementation—the one we presented in Section 2—is due to Rixner et al [30]. This algorithm was initially developed for single-threaded applications and shows good throughput performance in such scenarios. As shown in [23], however, it can have negative effects on fairness in chip-multiprocessor systems. The performance impact of different memory scheduling techniques in SMT processors and multiprocessors has been considered in [42, 22].

Fairness issues in managing access to shared resources have been studied in a variety of contexts. *Network fair queuing* has been studied in order to offer guaranteed service to simultaneous flows over a shared network link, e.g., [24, 40, 3], and techniques from network fair queuing have since been applied in numerous fields, e.g., CPU scheduling [6]. The best currently known algorithm for

network fair scheduling that also effectively solves the idleness problem was proposed in [2]. In [23], Nesbit et al. propose a fair memory scheduler that uses the definition of fairness in network queuing and is based on techniques from [3, 40]. As we pointed out in Section 4, directly mapping the definitions and techniques from network fair queuing to DRAM memory scheduling is problematic. Also, the scheduling algorithm in [23] can significantly suffer from the idleness problem. Fairness in *disk scheduling* has been studied in [4, 26]. The techniques used to achieve fairness in disk access are highly influenced by the physical association of data on the disk (cylinders, tracks, sectors, etc.) and can therefore not directly be applied in DRAM scheduling.

Shared hardware caches in multi-core systems have been studied extensively in recent years, e.g. in [35, 19, 14, 28, 9]. Suh et al. [35] and Kim et al. [19] develop hardware techniques to provide thread-fairness in shared caches. Fedorova et al. [9] and Suh et al. [35] propose modifications to the operating system scheduler to allow each thread its fair share of the cache. These solutions do not directly apply to DRAM memory controllers. However, the solution we examine in this paper has interactions with both the operating system scheduler and the fairness mechanisms used in shared caches, which we intend to examine in future work.

## 8 Conclusion

The advent of multi-core architectures has spurred a lot of excitement in recent years. It is widely regarded as the most promising direction towards increasing computer performance in the current era of power-consumption-limited processor design. In this paper, we show that this development—besides posing numerous challenges in fields like computer architecture, software engineering, or operating systems—bears important security risks.

In particular, we have shown that due to unfairness in the memory system of multi-core architectures, some ap-

plications can act as *memory performance hogs* and destroy the memory-related performance of other applications that run on different processors in the chip; without even being significantly slowed down themselves. In order to contain the potential of such attacks, we have proposed a memory request scheduling algorithm whose design is based on our novel definition of DRAM fairness. As the number of processors integrated on a single chip increases, and as multi-chip architectures become ubiquitous, the danger of memory performance hogs is bound to aggravate in the future and more sophisticated solutions may be required. We hope that this paper helps in raising awareness of the security issues involved in the rapid shift towards ever-larger multi-core architectures.

## Acknowledgments

We especially thank Burton Smith for continued inspiring discussions on this work. We also thank Hyesoon Kim, Chris Brumme, Mark Oskin, Rich Draves, Trishul Chilimbi, Dan Simon, John Dunagan, Yi-Min Wang, and the anonymous reviewers for their comments and suggestions on earlier drafts of this paper.

## References

- [1] Advanced Micro Devices. AMD Opteron. <http://www.amd.com/en/roce/productinformation/>.
- [2] J. H. Anderson, A. Block, and A. Srinivasan. Quick-release fair scheduling. In *RTSS*, 2003.
- [3] J. C. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. In *SIGCOMM*, 1996.
- [4] J. Bruno et al. Disk scheduling with quality of service guarantees. In *Proceedings of IEEE Conference on Multimedia Computing and Systems*, 1999.
- [5] A. Chander, J. C. Mitchell, and I. Shin. Mobile code security by Java bytecode instrumentation. In *DARPA Information Survivability Conference & Exposition*, 2001.
- [6] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *OSDI-4*, 2000.
- [7] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy*, 2006.
- [8] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *ISCA-26*, 1999.
- [9] A. Fedorova, M. Seltzer, and M. D. Smith. Cache-fair thread scheduling for multi-core processors. Technical Report TR-17-06, Harvard University, Oct. 2006.
- [10] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP*, 2003.
- [11] S. Gochman et al. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2), May 2003.
- [12] D. Grunwald and S. Ghiasi. Microarchitectural denial of service: Insuring microarchitectural fairness. In *MICRO-35*, 2002.
- [13] J. Hasan et al. Heat stroke: power-density-based denial of service in SMT. In *HPCA-11*, 2005.
- [14] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource. In *PACT-15*, 2006.
- [15] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *MICRO-37*, 2004.
- [16] Intel Corporation. Intel Develops Tera-Scale Research Chips. [http://www.intel.com/pre\\_room/archive/release/corp\\_b.htm](http://www.intel.com/pre_room/archive/release/corp_b.htm).
- [17] Intel Corporation. Pentium D. [http://www.intel.com/product/processor\\_number/chart/pentium\\_d.htm](http://www.intel.com/product/processor_number/chart/pentium_d.htm).
- [18] Intel Corporation. Terascale computing. <http://www.intel.com/research/platform/terascale/index.htm>.
- [19] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. *PACT-13*, 2004.
- [20] C. K. Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.
- [21] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. <http://www.c.virginia.edu/stream/>.
- [22] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI*, 2004.
- [23] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queueing memory systems. In *MICRO-39*, 2006.
- [24] A. K. Parekh. *A Generalized Processor Sharing Approach to Flow Control in Integrated Service Networks*. PhD thesis, MIT, 1992.
- [25] D. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In *11th USENIX Security Symposium*, 2002.
- [26] T. Pradhan and J. Haritsa. Efficient fair disk schedulers. In *3rd Conference on Advanced Computing*, 1995.
- [27] V. Prevelakis and D. Spinellis. Sandboxing applications. In *USENIX 2001 Technical Conf.: FreeNIX Track*, 2001.
- [28] N. Rafique et al. Architectural support for operating system-driven CMP cache management. In *PACT-15*, 2006.
- [29] S. Rixner. Memory controller optimizations for web servers. In *MICRO-37*, 2004.
- [30] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.
- [31] A. Rogers, M. C. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.
- [32] T. Sherwood et al. Automatically characterizing large scale program behavior. In *ASPLOS-X*, 2002.
- [33] E. Sprangle and O. Mutlu. Method and apparatus to control memory accesses. U.S. Patent 6,799,257, 2004.
- [34] Standard Performance Evaluation Corporation. *SPEC CPU2000*. <http://www.pec.org/cpu/>.
- [35] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. *HPCA-8*, 2002.
- [36] D. Wang et al. DRAMsim: A memory system simulator. *Computer Architecture News*, 33(4):100–107, 2005.
- [37] Y.-M. Wang et al. Checkpointing and its applications. In *FTCS-25*, 1995.
- [38] D. H. Woo and H.-H. S. Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, Feb. 2007.
- [39] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *ACM Computer Architecture News*, 23(1), 1995.
- [40] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. In *Proceedings of the IEEE*, 1995.
- [41] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *MICRO-33*, 2000.
- [42] Z. Zhu and Z. Zhang. A performance comparison of DRAM memory system optimizations for SMT processors. In *HPCA-11*, 2005.

# Binary Obfuscation Using Signals\*

Igor V. Popov, Saumya K. Debray, Gregory R. Andrews

*Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA*

*Email: {ipopov, debray, greg}@c .ari ona .edu*

## Abstract

Reverse engineering of software is the process of recovering higher-level structure and meaning from a lower-level program representation. It can be used for legitimate purposes—e.g., to recover source code that has been lost—but it is often used for nefarious purposes, e.g., to search for security vulnerabilities in binaries or to steal intellectual property. This paper addresses the problem of making it hard to reverse engineer binary programs by making it difficult to disassemble machine code statically. Binaries are obfuscated by changing many control transfers into signals (traps) and inserting dummy control transfers and “junk” instructions after the signals. The resulting code is still a correct program, but even the best current disassemblers are unable to disassemble 40%–60% of the instructions in the program. Furthermore, the disassemblers have a mistaken understanding of over half of the control flow edges. However, the obfuscated program necessarily executes more slowly than the original. Experimental results quantify the degree of obfuscation, stealth of the code, and effects on execution time and code size.

## 1 Introduction

Software is often distributed in binary form, without source code. Many groups have developed technology that enables one to reverse engineer binary programs and thereby reconstruct the actions and structure of the program. This is accomplished by disassembling machine code into assembly code and then possibly decompiling the assembly code into higher level representations [5, 6, 13]. While reverse-engineering technology has many legitimate uses (in particular, an important application of binary-level reverse engineering is to analyse malware in order to understand its behavior [4, 16–18, 25, 27, 33]), it can also be used to discover vulnerabilities, make unauthorized modifications, or steal intellectual property.

---

This work was supported in part by NSF Grants EIA-0080123, CCR-0113633, and CNS-0410918.

Since the first step in reverse engineering a binary is disassembly, many approaches to binary obfuscation focus on disrupting this step. This is typically done by identifying assumptions made by disassemblers, then transforming the program systematically so as to violate these assumptions without altering program functionality. Two fundamental assumptions made by disassemblers are that (1) the address where each instruction begins can be determined; and (2) control transfer instructions can be identified and their targets determined. The first assumption is used to identify the actual instructions to disassemble; most modern disassemblers use the second assumption to determine which memory regions get disassembled (see Section 2). In this context, this paper makes the following contributions:

1. It shows how the second of these assumptions can be violated, such that actual control transfers in the program cannot be identified by a static disassembler. This is done by replacing control transfer instructions—jumps, calls, and returns—by “ordinary” instructions whose execution raises traps at runtime; these traps are then fielded by signal handling code that carries out the appropriate control transfer. The effect is to replace control transfer instructions either with apparently innocuous arithmetic or memory operations or with what appear to be illegal instructions that suggest an erroneous disassembly.
2. It shows how the code resulting from this first transformation can be further obfuscated to additionally violate the first assumption stated above. This is done using a secondary transformation that inserts (unreachable) code, containing fake control transfers, after these trap-raising instructions, in order to make it hard to find the beginning of the true next instructions.

In earlier work, we showed how disassembly could be disrupted by violating the first assumption [20]; this paper extends that work by showing a different way to obfuscate binaries by replacing control transfer instructions

with apparently-innocuous non-control-transfer instructions. It is also very different from our earlier work on intrusion detection [21], which proposed a way to hinder certain kinds of mimicry attacks by obfuscating system call instructions. That work sought simply to disguise the instruction (`int $0x80` in Intel x86 processors) used by applications to trap into the OS kernel; more importantly, it required kernel modifications in order to work. By contrast, the work described in this paper applies to arbitrary control transfers in programs and requires no kernel modifications. Taken together, these two differences lead to significant differences between the two approaches in terms of goals, techniques, and effects.

It is important to note that code obfuscation is merely a technique: just as it can be used to protect software against attackers, so too it can be used to hide malicious content. The work presented here can therefore be seen from two perspectives: as a “defense model” of a new approach for protecting intellectual property, or as an “attack model” of a new approach for hiding malicious content. In either case, it goes well beyond current approaches to hiding the content of executable code. In particular, the obfuscations cause the best existing disassemblers to miss 40%–60% of the instructions in test programs and to make mistakes on over half of the control flow edges.

The remainder of the paper is organized as follows. Section 2 provides background information on static disassembly algorithms. Section 3 describes the new techniques for thwarting disassembly and explains how they are implemented. Section 4 describes how we evaluate the efficacy of our approach. Section 5 gives experimental results for programs in the SPECint-2000 benchmark suite. Section 6 describes related work, and Section 7 contains concluding remarks.

## 2 Disassembly Algorithms

This section summarizes the operation of disassemblers in order to provide the context needed to understand how our obfuscation techniques work. Broadly speaking, there are two approaches to disassembly: *static* and *dynamic*, the difference between them being that the former examines the program without execution, while the latter monitors the program’s execution (e.g., through a debugger) as part of the disassembly process. Static disassembly processes the entire input program all at once, while dynamic disassembly only disassembles those instructions that were executed for the particular input that was used. Moreover, with static disassembly it is easier to apply offline program analyses to reason about semantic aspects of the program under consideration. Finally, programs being disassembled statically are not

able to defend themselves against reverse engineering using anti-debugging techniques (see, for example, [2, 3]). For these reasons, static disassembly is a popular choice for low level reverse engineering. This paper focuses on static disassembly: its goal is to render static disassembly of programs sufficiently difficult and expensive as to force attackers to resort to dynamic approaches (which, in principle, can then be defended against).

There are two generally used techniques for static disassembly: *linear sweep* and *recursive traversal* [26]. The linear sweep algorithm begins disassembly at the input program’s first executable location, and simply sweeps through the entire text section disassembling each instruction as it is encountered. This method is used by programs such as the GNU utility `objdump` [24] as well as a number of link-time optimization tools [8, 23, 29]. The main weakness of linear sweep is that it is prone to disassembly errors resulting from the misinterpretation of data, such as jump tables, embedded in the instruction stream.

The recursive traversal algorithm uses the control flow instructions of the program being disassembled in order to determine what to disassemble. It starts with the program’s entry point, and disassembles the first basic block. When the algorithm encounters a control flow instruction, it determines the possible successors of that instruction—i.e., addresses where execution could continue—and proceeds with disassembly at those addresses. Variations on this basic approach to disassembly are used by a number of binary translation and optimization systems [6, 28, 30]. The main virtue of recursive traversal is that by following the control flow of a program, it is able to “go around” and thus avoid disassembly of data embedded in the text section. Its main weakness is that it depends on being able to determine the possible successors of each such instruction, which is difficult for indirect jumps and calls. The algorithm also depends on being able to find all the instructions that affect control flow.

A recently proposed generalization of recursive traversal is that of exhaustive disassembly [14, 15], which is the most sophisticated disassembly algorithm we are aware of. This approach aims to work around certain kinds of binary obfuscations by considering all possible disassemblies of each function. It examines the control transfer instructions in these alternative disassemblies to identify basic block boundaries, then uses a variety of heuristic and statistical reasoning to rule out alternatives that are unlikely or impossible. Like the recursive traversal algorithm it generalizes, the exhaustive algorithm thus also relies fundamentally on identifying and analyzing the behavior of control transfer instructions.

## 3 Signal-Based Obfuscation

### 3.1 Overview

In order to confuse a disassembler, we have to disrupt its notion of where the instructions are, what they are doing, and what the control flow is. The choices we have for altering the program are (1) changing instructions to others that produce the same result, and (2) adding instructions that do not have visible effects. Simple, local changes will obviously not confuse a disassembler or a human. More global and drastic changes are required.

The essential intuition of our approach can be illustrated via a simple example, given in Figure 1. The original code fragment on the left-hand side of the figure contains an unconditional jump to a location  $L$ ; the jump is preceded by *Code-before* and followed by *Code-after*. This code is obfuscated by replacing the jump by code that attempts to access an illegal memory location  $\ell$  and thereby generates a trap, which raises a signal. This is fielded by a handler that uses the address of the instruction that caused the trap to determine the target address  $L$  of the original jump instruction and to cause control to branch to  $L$ . In addition, *Bogus Code* is inserted after the trap point; this code appears to be reachable, but in fact it is not. Judicious choice of bogus code can throw off the disassembly even further.

This example illustrates a number of key aspects of our approach that increases the difficulty of statically de-obfuscating programs:

- A variety of different instructions and addresses can be used to raise a signal at runtime. The example uses a `load` from an illegal address, but we could have used many other alternatives, e.g., a `store` to a write-protected location, or a `load` from a read-protected location. Indeed, on an architecture such as the Intel x86, any instruction that can take a memory operand, including all the familiar arithmetic instructions, can be used for this purpose. Moreover, the address  $\ell$  used to generate the trap can be a legal address, albeit one that does not (at runtime) permit a particular kind of memory access. We can further hamper static reverse engineering by using something like an `mprotect` system call to (possibly temporarily) change the protection of the address  $\ell$  being used to generate the trap at runtime, so that it is not statically obvious that attempting a particular kind of memory access at address  $\ell$  will raise a trap.
- The address  $\ell$  used to generate the trap need not be a determinate value. For example, suppose that, as in typical 32-bit Linux systems, the top 1 GB of the virtual address space (i.e., addresses

to  $\text{0xffffffff}$ ) is reserved for the kernel, and is inaccessible to user processes. Then, any value of  $\ell$  in that address range will serve to generate the desired trap. Such values can be computed by starting with an arbitrary value and then using bit manipulations to obtain a value in the appropriate range, as shown below (one can imagine many variations on this theme), where  $A$  and  $B$  are arbitrary legal memory locations:

```
r0 := contents of A
r1 := contents of B
r1 := r1          /* r1's low byte ≥ 0 */
r1 := r1          /* r1 ≥ 0 */
r0 := r0 | r1
```

The actual runtime contents of memory locations  $A$  and  $B$  are unimportant here: the value computed into  $r_0$ —which may be different on different executions of this code—will nevertheless always point into protected kernel address space, causing memory accesses through  $r_0$  to generate a trap. Such indeterminacy can further complicate the task of reverse engineering the obfuscated code. Note that such indeterminate address computations can also be applied to generate an arbitrary address within a page (or pages) protected using `mprotect` as discussed above.

- A variety of different traps can be used. For example, in addition to the memory access traps mentioned above, we can use arithmetic exceptions, e.g., divide-by-zero. In fact, the “instruction” generating a trap need not be a legal instruction at all—i.e., we can use a byte pattern that does not correspond to any legal instruction to effect a control transfer via an illegal instruction trap. Such illegal byte sequences—which in general are indistinguishable from data legitimately embedded in the instruction stream—can be very effective in confusing disassemblers.
- The location following the trap-generating instruction is unreachable, but this is not evident from standard control flow analyses. We can exploit this by inserting additional “bogus” code after the trap-generating instruction to further confuse disassembly. Section 3.2 discusses this in more detail.

We could conceivably obfuscate every jump, call, or return in the source code. However, this would cause the program to execute *much* more slowly because of signal-processing overhead. We allow the user to specify a hot-code threshold, and we only obfuscate control transfers that are not in hot parts of the original program (see Section 5 for details). Even so, we are able to obfuscate about a third of the instructions in hot code blocks.

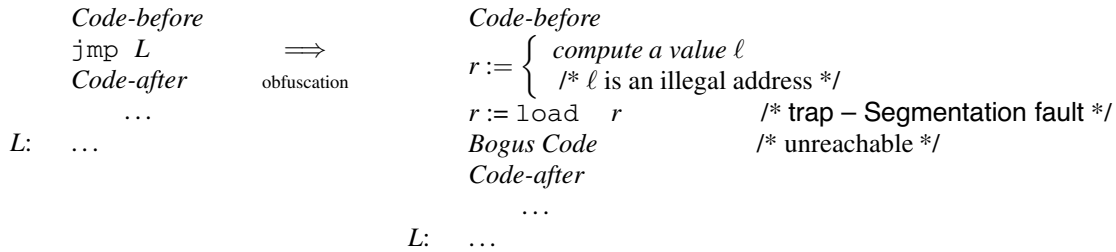


Figure 1: A Simple Example of our Approach to Obfuscation

Before obfuscating a program, we first instrument the program to gather edge profiles, and then we run the instrumented version on a training input. The obfuscation process itself has several steps. First, using the profile data and hot-cold threshold, determine which control transfers should be obfuscated and modify each such instruction as shown in Figure 1. Second, insert bogus code at unreachable code locations such as after trap-generating instructions. Third, intersperse signal handling and restore (return from signal) code with the original program code. Fourth, compute the new memory layout, construct a table of mappings from trap instructions to target addresses, and patch the restore code to use this table via a perfect hash function. Finally, assemble a new, obfuscated binary.

### 3.2 Program Obfuscations

Within our obfuscator, the original program is represented as an interprocedural control-flow graph (ICFG). The nodes are basic blocks of machine instructions; the edges represent the control flow in the program.

#### Obfuscating Control Transfers

After some initialization actions, our obfuscator makes one pass through the original program to *flip* conditional branches—i.e., reverse the sense of the branch condition and insert an explicit unconditional jump after it to maintain the program’s semantics. This transformation has the effect of increasing the set of candidate locations where our obfuscation can be applied. Our obfuscator then makes a second pass through the program to find and modify all control transfer instructions that are to be obfuscated.

To obfuscate a control transfer instruction, we insert Setup code that prepares for raising a signal and then Trap code that causes a signal. The Setup code (1) allocates space on the stack for use by the signal handler to store the address of the trap instruction, and (2) sets a flag that indicate to the signal handler that the coming signal is from obfuscated code, not the original pro-

gram itself. To set a flag, we use a pre-allocated array (initialized to zero), and the Setup code moves a random non-zero value into a randomly chosen element of the array. Jump, return, and call instructions are obfuscated in nearly identical ways; the only essential difference is the amount of stack space that we need to allocate in order to effect the intended control transfer. The Trap code generates a trap, which in turn raises a signal. In order not to interfere with signal handlers that might be installed in the original program, we only raise signals for which the default action is to dump core and terminate the program. In particular we use illegal instruction (SIGILL), floating point exception (SIGFPE), and segmentation violation (SIGSEGV).

To determine which kind of trap to raise—and to avoid the need to save and later restore program registers—we first do liveness analysis to determine which registers are live at the trap point and which are available for us to use. If no register is available, we randomly generate an illegal instruction from among several possible choices. Otherwise, we generate code to load a zero into a free register  $r$ , then either dereference  $r$  (to cause a segmentation fault), or divide by  $r$  (to cause a floating point exception). Since indirect loads are far more frequent than divides in real programs, most of the time we choose the former.

If we simply moved a zero into a register each time that we wanted to trigger a floating point exception or segmentation fault, there would be dozens of such instructions that would be a signature for our obfuscation. To avoid this, we generate a sequence of instructions by using multiple, randomly chosen rewriting rules that perform value-preserving transformations on the registers that are free at each obfuscation point. Appendix A describes how we randomize the computation of values.

#### Inserting Bogus Code

After obfuscating a control transfer instruction, we next insert bogus code—a conditional branch and some junk bytes—to further confuse disassemblers. This is shown in Figure 2. Since the trap instruction has the effect of

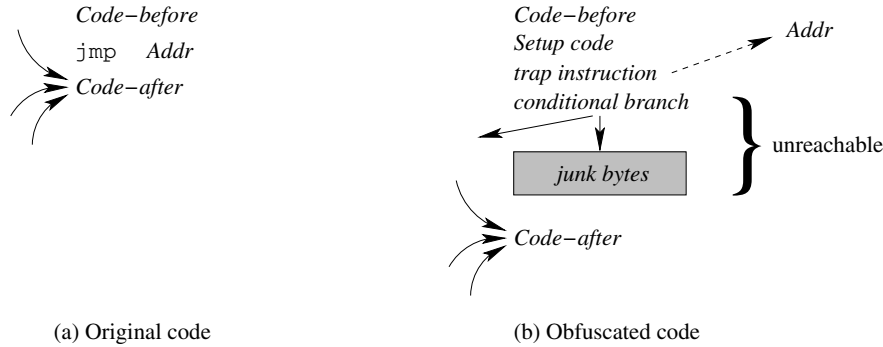


Figure 2: Bogus code insertion

an unconditional control transfer, the conditional branch immediately following the trap is “bogus code” that will not be reachable in the obfuscated program, and hence it will not be executed. The purpose of adding this instruction is to confuse the control flow analysis of the program by misleading the disassembler into identifying a spurious edge in the control flow graph; the control flow edge so introduced can also lead to further disassembly errors at the target of this control transfer. A secondary benefit of such bogus conditional branches is that they help improve the stealthiness of the obfuscation, since otherwise the disassembly would produce what appeared to be long sequences of straight-line code without any branches, which would not resemble code commonly encountered in practice. We randomly select an unconditional branch—based on how frequently the different kinds occur in normal programs—and use a random PC-relative displacement.

The junk bytes are a proper prefix of a legal instruction. The goal is to cause a disassembler to consume the first few bytes of *Code-after* when it completes the instruction that starts with the junk bytes. This will ideally cause it to continue to misidentify the true instruction boundaries for at least a while.<sup>1</sup> We determine the prefix length  $n$  that maximizes the disassembly error for subsequent instructions ( $n$  depends only on the instructions in *Code-after*), and insert the first  $n$  bytes of an instruction chosen randomly from a number of different alternatives.

### Building the Mapping Table

After obfuscating control flow and inserting bogus code, our obfuscator computes a memory layout for the obfuscated program and determines final memory addresses.

<sup>1</sup>This technique only works on variable-instruction-length architectures such as the IA-32. Moreover, disassemblers tend to resynchronize relatively quickly, so that on average they are confused for only three or four instructions before again finding the true instruction boundaries.

Among these are the addresses of all the trap instructions that have been inserted. The obfuscator then goes through the control flow graph and gathers the information it needs to build a table that maps trap locations to original targets.

Suppose that  $N$  control transfer instructions have been obfuscated. Then there are  $N$  rows in the mapping table, one for each trap point. Each row contains a flag that indicates the type of transfer that was replaced, and zero, one, or two target addresses, depending on the value of the flag. To make it hard to reverse engineer the contents and use of this table, we use two techniques. First, we generate a perfect hash function that maps the  $N$  trap addresses to distinct integers from 0 to  $N - 1$  [12], and we use this function to get indices into the mapping table; this machine code is quite inscrutable and hence hard to reverse engineer. Second, to make it hard to discover the target addresses in the mapping table, in place of each target address  $T$  we store a value  $XT$  that is the XOR of  $T$  and the corresponding trap address  $S$ .

### 3.3 Signal Handling

When an instruction raises a signal, the processor stores its address  $S$  on the stack, then traps into the kernel. Figure 3(a) shows the components and control transfers that normally occur when a program raises a signal at address  $S$  and has installed a signal handler that returns back to the program at the same address. (If no handler has been installed, the kernel takes the default action for the signal.) Figure 3(b) shows the components and control transfers that occur in our implementation. The essential differences are that we return control to a different target address  $T$ , and we do so by causing the kernel to transfer control to our restore code rather than back to the trap address. We allow obfuscated programs to install their own signal handlers, as described below.

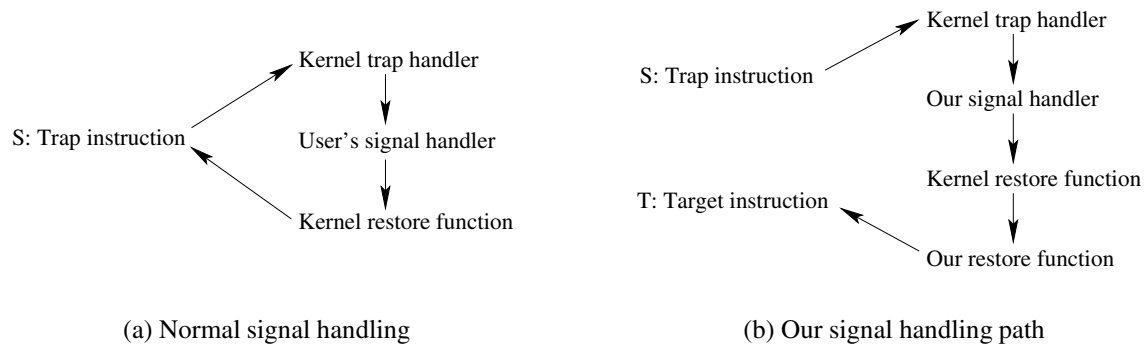


Figure 3: Signal Handling: Normal and Obfuscated Cases

### Handler and Restore Code Actions

We trigger the path shown in Figure 3(b) when a signal is raised from a trap location that we inserted in the binary. However, other instructions in the original program might raise the illegal instruction, floating point exception, or segmentation fault signals. To tell the difference, we use a global array that is initialized to zero. In the Setup code before each of the traps we insert in the program, we set a random element of this array to a non-zero value. In our signal handler, we loop through this array to see if any value is nonzero (and we then reset it to zero).

In the normal case where our signal handler is processing one of the traps we inserted in the program, it overwrites the kernel restore function's return address with the address of our restoration code. That code (1) invokes the perfect hash function on the trap address *S* (which was put in the stack space allocated by our signal handler), (2) looks up the original target address, (3) resets the stack frame as appropriate for the type of control transfer and (4) transfers control (via a *return* instruction) to the original target address.

To make it harder for an attacker to find and reverse engineer the signal handler, we disperse our handler and restore code over the program, i.e., we split the code into multiple basic blocks and interleave these in with the original program code. We also make multiple slightly different copies of each code block so that we are not always using the same locations each time we handle a signal. As will be shown in Section 5, we are able to obfuscate many hot instructions as a side effect of obfuscating cold code. These include some of the code we introduce to handle signals.

### Interaction With Other Signals

We allow the original program to install signal handlers and dynamically to change signal handling semantics. By analyzing the binary, we determine whether it in-

stalls signal handlers: this is done by checking to see whether there are any calls to system library routines (e.g., `signal`) that install signal handlers. We transform the code to intercept these calls at runtime and record, in a table, the signals that are being handled and the address of the corresponding signal handler. When our signal handler determines that a signal did not get raised by one of our obfuscations (by examining the array of flags), it consults this table. If the user installed a handler, we call that handler then return to the original program. Otherwise, we take the default action for that kind of signal.

Although in general we are able to handle interactions between signal handling in our code and the original program, we discovered one instance of a race condition. In particular, one of the SPECint-95 benchmark programs, *m88ksim*, installs a handler for SIGINT, the interrupt signal. If we obfuscate that program, run the code, and interrupt the program while it happens to be in our handler, the program will cause a segmentation fault and crash. To solve this type of problem, our signal handler needs to delay the processing of other signals that might be raised. (On Unix this can be done by having the signal handler call the `igprocma` function, or by using `igaction` when we (re)install the handler.) Once our trap processing code gets back to the restore code block of the obfuscated program, it can safely be interrupted because it is through manipulating kernel addresses. However, our current implementation does not yet block other signals.

An even worse problem would occur in a multi-threaded program, because multiple traps could occur and have to be handled at the same time. Signal handling is not thread safe in general in Unix systems, so our obfuscation method cannot be used in an arbitrary multithreaded program. However, this is a limitation of Unix, not our method.

### 3.4 Attack Scenarios

Recall that our goal is to make static disassembly difficult enough to force any adversary to resort to dynamic techniques. Here we discuss why we believe our scheme is able to attain this goal.

We assume that our approach is known to the adversary. As discussed in Section 3.2, the specifics of the obfuscation as applied to a particular program—the setup code, the kind of trap used for any particular control transfer, the code sequence used to generate traps, as well as the bogus control transfers inserted after the trap instruction—are chosen randomly. This makes it difficult for an adversary to identify the location of trap instructions and bogus control transfers simply by inspecting the obfuscated code.

Since locating the obfuscation code by simple inspection is not feasible, the only other possibility to consider, for statically identifying the obfuscation instructions, is static analysis. This is difficult for two reasons. The first is the sheer number of candidates: for example, in principle any memory operation can raise an exception and is therefore potentially a candidate for analysis. Secondly, the problem of statically determining the values of the operands of such candidate instructions is difficult, both theoretically [22] and in practice, especially because, as discussed in Section 3.1, such operands need not be fixed constant values. Furthermore, if a byte sequence is encountered in the disassembly that does not encode a legal instruction (and therefore cannot be subjected to static analysis), it can be either a part of the obfuscation (i.e., is “executed” and causes a trap), or it can be data embedded in the instruction stream: determining which of these is the case is in general an undecidable problem.

## 4 Evaluation

We measure the efficacy of obfuscation in two ways: by the extent of incorrect disassembly of the input, and by the extent of errors in control flow analysis of the disassembled input. These quantities are related, in the sense that an incorrect disassembly of a control transfer instruction will result in a corresponding error in the control flow graph obtained for the program. However, it is possible, in principle, to have a perfect disassembly and yet have errors in control flow analysis because control transfer instructions have been disguised as innocuous arithmetic instructions or bogus control transfers have been inserted.

### 4.1 Evaluating Disassembly Errors

We measure the extent of disassembly errors using a measure we call the *confusion factor* for the instructions, basic blocks, and functions. Intuitively, the confusion factor measures the fraction of program units (instructions, basic blocks, or functions) in the obfuscated code that were incorrectly identified by a disassembler. More formally, let  $A$  be the set of all *actual* instruction addresses, i.e., those that would be encountered when the program is executed, and let  $P$  be the set of all *perceived* instruction addresses, i.e., those addresses produced by a static disassembly. Then  $A - P$  is the set of addresses that are not correctly identified as instruction addresses by the disassembler. We define the *confusion factor*  $CF$  to be the fraction of instruction addresses that the disassembler fails to identify correctly:<sup>2</sup>

$$CF = |A - P|/|A|.$$

Confusion factors for functions and basic blocks are calculated analogously: a basic block or function is counted as being “incorrectly disassembled” if any of the instructions in it is incorrectly disassembled. The reason for computing confusion factors for basic blocks and functions as well as for instructions is to determine whether the errors in disassembling instructions are clustered in a small region of the code, or whether they are distributed over significant portions of the program.

### 4.2 Evaluating Control Flow Errors

Two kinds of errors can occur when comparing the control flow structure of the disassembled program  $P_{disasm}$  with that of the original program  $P_{orig}$ . First,  $P_{disasm}$  may contain some edge that does not appear in  $P_{orig}$ , i.e., the disassembler may mistakenly find a control flow edge where the original program did not have one. Second,  $P_{disasm}$  may not contain some edge that appears in  $P_{orig}$ , i.e., the disassembler may fail to find an edge that was present in the original program. We term the first kind of error *overestimation errors* (written  $\Delta_{over}$ ) and the second kind *underestimation errors* (written  $\Delta_{under}$ ), and express them relative to the number of edges in the original program. Let  $E_{orig}$  be the set of control flow edges in the original program and  $E_{disasm}$  the set of control flow edges identified by the disassembler, then:

$$\begin{aligned}\Delta_{over} &= |E_{disasm} - E_{orig}|/|E_{orig}| \\ \Delta_{under} &= |E_{orig} - E_{disasm}|/|E_{orig}|\end{aligned}$$

---

<sup>2</sup>We also considered taking into account the set  $P - A$  of addresses that are erroneously identified as instruction addresses by the disassembler, but we rejected this approach because it “double counts” the effects of disassembly errors.

Even if we assume a perfect “attack disassembler” that does not incur any disassembly errors, its output will nevertheless contain control flow errors arising from two sources. First, it will fail to identify control transfers that have been transformed to trap-raising instructions. Second, it will erroneously identify bogus control transfers introduced by the obfuscator. We can use this to bound the control flow errors even for a perfect disassembly. Suppose that  $n_{trap}$  control flow edges are lost from a program due to control transfer instructions being converted to traps, and  $n_{bogus}$  bogus control flow edges are added by the obfuscator. Then, a lower bound on underestimation errors,  $\min \Delta_{under}$ , is obtained when the only control transfers that the attack disassembler fails to find are those that were lost due to conversion to trap instructions:  $\min \Delta_{under} = n_{trap}/E_{orig}$ . An upper bound on overestimation errors,  $\max \Delta_{over}$ , is obtained when every bogus conditional branch inserted by the obfuscator is reported by the disassembler:  $\max \Delta_{over} = n_{bogus}/E_{orig}$ .

## 5 Experimental Results

We evaluated the efficacy of our techniques using eleven programs from the SPECint-2000 benchmark suite.<sup>3</sup> Our experiments were run on an otherwise unloaded 2.4 GHz Pentium IV system with 1 GB of main memory running RedHat Linux (Fedora Core 3). The programs were compiled with *gcc* version 3.4.4 at optimization level `-O3`. The programs were profiled using the SPEC training inputs and these profiles were used to identify any hot spots during our transformations. The final performance of the transformed programs was then evaluated using the SPEC reference inputs. Each execution time reported was derived by running seven trials, removing the highest and lowest times from the sampling, and averaging the remaining five.

We experimented with three different “attack disassemblers” to evaluate our techniques: GNU *objdump* [24]; IDA Pro [11], a commercially available disassembly tool that is generally regarded to be among the best disassemblers available;<sup>4</sup> and an exhaustive disassembler by Kruegel *et al.* that was engineered to handle obfuscated binaries [15]. *objdump* uses a straightforward linear sweep algorithm, while IDA Pro uses recursive traversal. The exhaustive disassembler of Kruegel *et al.* takes into account the possibility that the input binary may be obfuscated by not making any assumptions about instruction boundaries. Instead, it considers alternative disassemblies starting at every byte in the code region of the program, then examines these alternatives

using a variety of statistical and heuristic analyses to discard those that are unlikely or impossible. Kruegel *et al.* report that this approach yields significantly better disassemblies on obfuscated inputs than other existing disassemblers [15]; to our knowledge, the exhaustive disassembler is the most sophisticated disassembler currently available.

In order to maintain a reasonable balance between the extent of obfuscation and the concomitant runtime overhead, we obfuscated only the “cold code” in the program—where a basic block is considered “cold” if, according to the execution profiles used, it is not executed. We evaluated a number of different combinations of obfuscations. The data presented below correspond to the combination that gave the highest confusion factors without excessive performance overhead: flip branches to increase the number of unconditional jumps in the code (see Section 3.2); convert all unconditional control transfers (jumps, calls, and function returns) in cold code to traps; insert bogus code after traps; and insert junk bytes after *jmp*, *ret*, and *halt* instructions.

### Disassembly Error

The extent of disassembly error, as measured by confusion factors (Section 4.1) is shown in Figure 4(a). The results differ depending on the attack disassembler, but the results for each disassembler are remarkably consistent across the benchmark programs. Because we have focused primarily on disguising control transfer instructions by transforming them into signal-raising instructions, it does not come as a surprise that the straightforward linear sweep algorithm used by the *objdump* disassembler has the least confusion at 43% of the instructions on average. However, these are spread across 68% of the basic blocks and 90% of the functions. The other disassemblers are confused to a much greater extent—55% for the exhaustive disassembler and 57% for IDA Pro, on average—but these are more somewhat more clustered as they cover only about 60% of the basic blocks and slightly fewer functions (89% and 85%, respectively).

Overall, the instruction confusion factors show that a significant portion of each binary is disassembled incorrectly; the basic block and function confusion factors show that the errors in disassembly are distributed over most of the program. Taken together, these data show that our techniques are effective even against state-of-the-art disassembly tools.

We have also measured the relative confusion factors for hot and cold instructions, i.e., those in hot versus cold basic blocks. For *objdump*, the confusion factors are nearly identical at 42% of the hot instructions and 44% of

<sup>3</sup>We did not use the *eon* programs from this benchmark suite because we were not able to build it.

<sup>4</sup>We used IDA Pro version 4.3 for the results reported here.

the cold instructions (again on average). The exhaustive disassembler was confused by fewer of the hot instructions (35%) but more of the cold instructions (59%). IDA Pro did the best on hot instructions at 28% confusion, on average, but worst on cold instructions at 62% confusion. It is not surprising that Kruegel and IDA Pro did better with hot code, because we did not obfuscate it except to insert junk after hot unconditional jumps, and junk by itself should not confuse an exhaustive or recursive descent disassembler. Then again, these disassemblers still failed to disassemble about a third of the hot code.

As an aside, we had thought that interleaving hot and cold basic blocks would cause more of the obfuscations in cold code to cause disassembly errors to “spill over” into succeeding hot code and increase the confusion there. This turns out to be the case for *objdump*, which is especially confused by junk byte insertion. However, IDA Pro and the exhaustive disassembler are still able to find most hot code blocks. In fact, such interleaving introduces additional unconditional jumps in the code, e.g., from one hot block to the next one, jumping around the intervening cold code. The exhaustive disassembler and IDA Pro are able to find these jumps and use them to improve disassembly, resulting in less confusion when hot and cold code are interleaved. Moreover, programs run more slowly when hot and cold blocks are interleaved due to poorer cache utilization.

## Control Flow Obfuscation

Figure 4(b) shows the effect of our transformations in obfuscating the control flow graph of the program. The second column gives, for each program, the actual number of control flow edges in the original program. These are counted as follows: each conditional branch gives rise to two control flow edges; each unconditional branch (direct or indirect) gives rise to a single edge; and each function call gives two control flow edges—one corresponding to a “call edge” to the callee’s entry point, the other to a “return edge” from the callee back to the caller. Column 3 gives the number of control flow edges removed due to the conversion of control flow instructions to traps, while column 4 gives the number of bogus control flow edges added by the obfuscator. Columns 5 and 6 give, respectively, an upper bound on the overestimation error and a lower bound on the underestimation error. The remaining columns give, for each attack disassembler, the extent to which it incurs errors in constructing the control flow graph of the program, as discussed in Section 4.2.

It can be seen from Figure 4(b) that none of the three attack disassemblers tested fares very well at constructing the control flow graph of the program. *bjdump* fails to find over 63% of the control flow edges in the

PROGRAM	EXECUTION TIME (SECS)		
	Original ( $T_0$ )	Obfuscated ( $T_1$ )	Slowdown ( $T_1/T_0$ )
<i>bzip2</i>	283.011	377.620	1.334
<i>crafty</i>	140.741	1222.992	1.584
<i>gap</i>	146.367	152.673	1.043
<i>gcc</i>	151.624	247.552	1.633
<i>gzip</i>	210.036	209.502	0.997
<i>mcf</i>	425.971	427.132	1.003
<i>parser</i>	301.079	302.040	1.003
<i>perlbmk</i>	220.851	461.828	2.091
<i>twolf</i>	569.163	586.259	1.030
<i>vortex</i>	235.649	240.648	1.021
<i>vpr</i>	319.475	328.563	1.028
GEOM. MEAN			1.210

Figure 5: Effect of Obfuscation on Execution Speed

original program; at the same time, it reports over 71% spurious edges (relative to the number of original edges in the program) that are not actually present in the program. The exhaustive disassembler fails to find over 60% of the edges in the original program, and reports over 27% spurious edges. IDA Pro fails to find over 63% of the control flow edges in the original program and reports over 41% spurious edges. Again the results for each disassembler are very consistent across the benchmark programs.

Also significant are the error bounds reported in columns 5 and 6 of Figure 4(b). These numbers indicate that, even if we suppose perfect disassembly, the result would incur up to 85.5% overestimation error and at least 28.93% underestimation error.

## Execution Speed

Figure 5 shows the effect of obfuscation on execution speed. For some programs—such as *gap*, *gzip*, *mcf*, *parser*, *twolf*, *vortex*, and *vpr*—the execution characteristics on profiling input(s) closely match those on the reference input, so there is essentially no slowdown. (In fact, *gzip* ran faster after obfuscation; we believe this is due to a combination of cache effects and experimental errors resulting from clock granularity.) For other programs—such as *crafty*, *gcc* and *perlbmk*—the profiling inputs are not as good predictors of the runtime characteristics of the program on the reference inputs, and this results in significant slowdowns: a factor of 1.6 for *crafty* and *gcc* and 2.1 for *perlbmk*. The mean slowdown seen for all eleven benchmarks is 21%.

We also measured the effect on execution speed of obfuscating a portion of the hot code blocks. Let  $\theta$  specify the fraction of the total number of instructions ex-

PROGRAM	OBJDUMP			EXHAUSTIVE			IDA PRO		
	<i>Instrs</i>	<i>Blocks</i>	<i>Funcs</i>	<i>Instrs</i>	<i>Blocks</i>	<i>Funcs</i>	<i>Instrs</i>	<i>Blocks</i>	<i>Funcs</i>
<i>bzip2</i>	44.19	69.62	89.59	55.57	60.29	89.33	59.88	62.94	85.99
<i>crafty</i>	41.09	68.70	90.26	55.94	61.04	87.92	54.22	59.26	84.71
<i>gap</i>	42.98	66.78	90.19	52.64	56.13	87.04	55.70	57.92	83.77
<i>gcc</i>	46.32	68.67	89.26	55.89	58.24	87.58	54.67	55.49	82.28
<i>gzip</i>	44.26	69.56	90.25	53.61	58.91	87.13	61.65	63.45	85.18
<i>mcf</i>	44.85	69.91	89.20	57.32	60.53	87.80	58.68	61.27	84.85
<i>parser</i>	44.28	68.83	91.70	55.19	59.40	88.87	57.85	61.27	85.06
<i>perlbmk</i>	45.34	69.08	89.80	55.62	58.82	90.09	55.16	56.14	85.89
<i>twolf</i>	41.90	68.32	89.03	56.29	61.63	88.80	57.77	61.74	84.77
<i>vortex</i>	39.80	69.40	93.28	58.05	65.77	93.10	55.96	64.04	90.98
<i>vpr</i>	42.31	68.19	86.67	54.20	59.91	87.49	59.01	63.27	82.16
GEOM. MEAN:	43.35	68.82	89.92	55.46	60.02	88.63	57.28	60.55	85.03

(a) Disassembly Errors (Confusion Factor, %)

PROGRAM	$E_{orig}$	$n_{trap}$	$n_{bogus}$	max $\Delta_{over}$	min $\Delta_{under}$	OBJDUMP		EXHAUSTIVE		IDA PRO	
						$\Delta_{over}$	$\Delta_{under}$	$\Delta_{over}$	$\Delta_{under}$	$\Delta_{over}$	$\Delta_{under}$
<i>bzip2</i>	47933	13933	42236	88.11	29.07	72.60	64.82	28.31	61.40	40.44	65.66
<i>crafty</i>	59507	17243	50868	85.48	28.98	72.26	62.17	25.28	60.71	43.12	62.20
<i>gap</i>	98793	26603	82374	83.38	26.93	69.94	60.90	25.35	57.20	41.49	60.84
<i>gcc</i>	237491	67818	193570	81.51	28.56	67.18	63.98	25.41	58.36	41.93	59.32
<i>gzip</i>	48467	13931	42722	88.15	28.74	72.56	64.56	29.64	59.64	38.34	66.09
<i>mcf</i>	43376	12329	38220	88.11	28.42	72.57	65.32	26.13	61.46	42.79	63.74
<i>parser</i>	59823	16688	50858	85.01	27.90	70.65	63.94	27.77	59.78	40.87	63.46
<i>perlbmk</i>	116711	33748	100298	85.94	28.92	70.67	64.75	26.99	59.31	43.27	59.66
<i>twolf</i>	62210	18061	52916	85.06	29.03	71.90	62.27	28.55	61.48	40.85	63.94
<i>vortex</i>	97242	32507	81734	84.05	33.43	72.37	63.29	30.28	65.25	41.62	66.18
<i>vpr</i>	55187	15811	47414	85.92	28.65	71.76	61.92	27.92	59.76	38.60	65.56
GEOM. MEAN:				85.50	28.93	71.30	63.43	27.37	60.36	41.18	63.28

(b) Control Flow Errors (%)

**Key:**  $E_{orig}$ : edges in original program  
 $n_{trap}$ : control flow edges lost due to trap conversion  
 $n_{bogus}$ : bogus control flow edges added  
max  $\Delta_{over}$ : upper bound on overestimation errors  
min  $\Delta_{under}$ : lower bound on underestimation errors

Figure 4: Efficacy of obfuscation

ecuted at runtime that should be accounted for by hot basic blocks. (The execution times in Figure 5 are for  $\theta = 1.0$ , i.e., all basic blocks with an execution count greater than 0 are consider hot.) If we run our obfuscator with  $\theta = 0.999$ —which means that, in addition to cold basic blocks, we obfuscate the hot basic blocks that account for just a tenth of a percent of the dynamic execution count—then the mean slowdown for the eleven benchmarks increases to 2.38. For smaller values of  $\theta$ , the situation is far worse: at  $\theta = 0.99$  the mean slowdown is 6.79, and at  $\theta = 0.9$  the mean slowdown climbs to 43.39. The confusion factor increases somewhat when  $\theta$  is decreased, but even at  $\theta = 0.9$  the increase in confusion is less than 10% relative to the confusion at  $\theta = 1.0$ .

## Program Size

Figure 6 shows the impact of obfuscation on the size of the text and initialized data sections. It can be seen that the size of the text section increases by factors ranging from 1.90 (*crafty*) to almost 2.1 (*vortex*), with a mean increase of a factor of 2.01. The relative growth in the size of the initialized data section is considerably larger, ranging from a factor of about 10 (*crafty*) to a factor of over 58 (*twolf*), with a mean growth of a factor of 26.46. The growth in the size of the initialized data is due to the addition of the mapping tables used to compute the type of each branch as well as its target address. However, this large relative growth in the data section size is due mainly to the fact that the initial size of this section is

PROGRAM	TEXT SECTION (KB)			INITIALIZED DATA SECTION (KB)			COMBINED: TEXT+DATA (KB)		
	Original ( $T_0$ )	Obfusc. ( $T_1$ )	Change ( $T_1/T_0$ )	Original ( $D_0$ )	Obfusc. ( $D_1$ )	Change ( $D_1/D_0$ )	Original ( $C_0$ )	Obfusc. ( $C_1$ )	Change ( $C_1/C_0$ )
<i>bzip2</i>	343.6	694.2	2.02	6.4	145.4	22.64	350.0	840.0	2.40
<i>crafty</i>	474.5	903.1	1.90	19.7	192.4	9.78	494.2	1,095.5	2.22
<i>gap</i>	690.9	1,351.6	1.96	6.8	273.1	39.95	697.8	1,624.7	2.33
<i>gcc</i>	1,494.4	3,051.8	2.04	21.9	675.5	30.88	1,516.3	3,727.3	2.46
<i>gzip</i>	344.9	699.8	2.03	5.8	145.0	24.94	350.7	844.8	2.41
<i>mcf</i>	301.6	618.8	2.05	3.3	128.2	39.03	304.9	747.0	2.45
<i>parser</i>	402.6	833.1	2.07	5.7	180.0	31.32	408.3	1,013.f	2.48
<i>perlbnk</i>	808.6	1,612.2	1.99	32.8	359.0	10.95	841.4	1,971.3	2.34
<i>twolf</i>	472.f	930.9	1.97	3.3	192.2	58.46	475.7	1,123.2	2.36
<i>vortex</i>	725.3	1,513.5	2.09	19.8	340.7	17.23	745.1	1,854.2	2.49
<i>vpr</i>	407.1	800.7	1.97	3.4	165.5	48.45	410.5	966.2	2.35
GEOM. MEAN			2.01			26.46			2.39

Figure 6: Effect of Obfuscation on Text and Data Section Sizes

not very large. When we consider the total increase in memory requirements due to our technique, obtained as the sum of the text and initialized data sections, we see that it ranges from a factor of 2.22 (*crafty*) to about 2.5 (*parser* and *vortex*), with a mean growth of a factor of about 2.4.

The increase in the size of the text section arises from three sources. The first of these is the code required to set up and raise the trap for each obfuscated control transfer instruction. The second is the junk bytes and bogus conditional branch inserted after a trap instruction. Finally, there is the signal handler and restore code. In our current implementation, the first two of these sources—the setup code for a trap and bogus code inserted after a trap—introduces on average an additional 30 bytes of memory for each obfuscated control transfer instruction. This accounts for over 95% of the total increase in the text section size. Each obfuscated control transfer also adds three memory words (12 bytes) to the initialized data section, accounting for the increase in the size of this section.

## Stealth

The *stealth* of an obfuscation refers to the difficulty of distinguishing code introduced by an obfuscator from the original program code. We estimate the stealth of our obfuscations by comparing the distributions of (i) opcodes and (ii) opcode pairs from adjacent instructions. The points of comparison are the original program, the obfuscated program (for which we know the actual instructions), and the instructions observed by the different attack disassemblers we tested. The idea is to see whether the obfuscation process introduces any atypical instruction sequence signatures that could be used to identify the obfuscation code statically.

Figure 7 shows the distribution of individual opcodes. Because of the large number of opcodes in the x86 architecture, we have grouped the commonly used opcodes into 27 sets—moves, conditional jumps, and so on—and also added sets for illegal instructions and all other instructions. The 29 different sets of opcodes are given along the x-axis (“jc” denotes conditional jumps), while the y-axis shows the percentage of instructions in any program accounted for by that opcode. The y-axis uses a logarithmic scale to in order to magnify the data ranges for small percentage values. Each data point was obtained by computing the opcode distribution for each of the SPECint-2000 benchmarks in our test suite: the original binary, the distribution in the obfuscated program obtained from the obfuscator, and the distributions for the disassemblies produced by each of the attack disassemblers. We then used these data to compute the arithmetic mean, minimum, and maximum values of instruction percentages for that opcode across all of our test programs. For each opcode, the dark line in Figure 7 shows the mean instruction percentage for each opcode, together with the range of its values. This line is then followed immediately to the right by a set of lighter lines that show the values for the obfuscated programs as well as those in the disassemblies obtained from the attack disassemblers.

Figure 7 illustrates that, in most cases, the mean value of each opcode’s range in the obfuscated code is within the range of values in the unobfuscated benchmark code. Calls, returns, and jumps are somewhat less frequent for the obvious reason that we obfuscated many of those instructions. Conditional jumps are somewhat more frequent because we added these to bogus code. On balance, however, there are no obvious outliers that an attacker could use to use as a signature for where obfuscations occur.

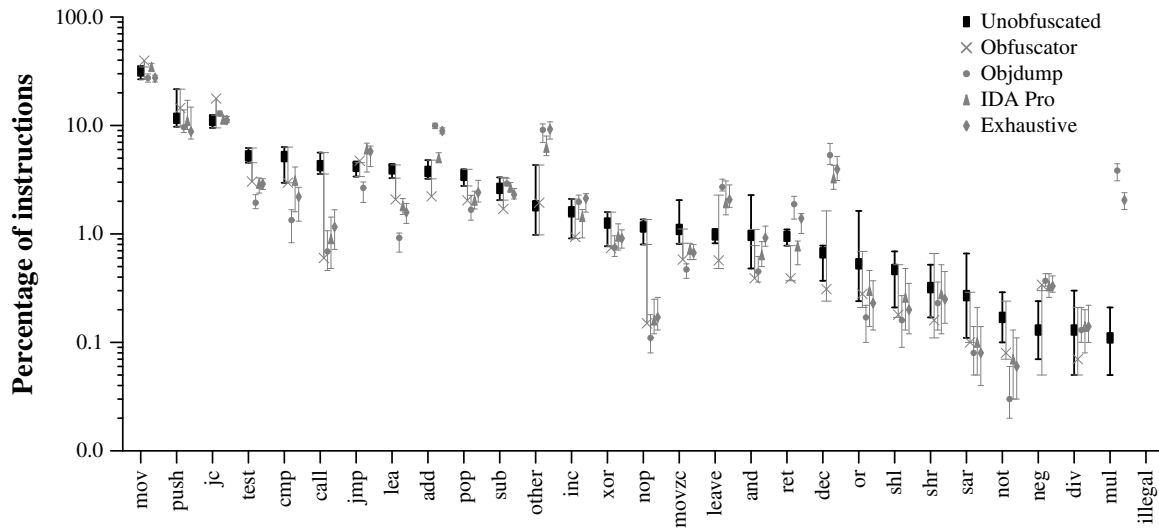


Figure 7: Obfuscation Stealth I: Distribution of Individual Opcodes

Figure 8 shows the distribution of pairs of adjacent opcodes, not including pairs involving illegal opcodes. Due to space constraints, we show the data for just one attack disassembler, IDA Pro; the data are generally similar for the other attack disassemblers. Figure 8(a) shows the actual distribution of opcode pairs in the obfuscated code, while Figure 8(b) shows the distribution for the disassembly obtained from IDA Pro. To reduce visual clutter in these figures, we plot the ranges of values for each opcode pair in the unobfuscated code (the dark band running down the graph), but only the mean values for the obfuscated code.

There are two broad conclusions to be drawn from Figure 8. First, as can be seen from Figure 8(a), the actual distribution of adjacent opcode pairs in the obfuscated code is, by and large, reasonably close to that of the original code; however, there are a few opcode pairs, very often involving conditional jumps, that occur with disproportionate frequency. The selection of obfuscation code to eliminate such atypical situations is an area of future work. The second conclusion is that, as indicated by Figure 8(b), the opcode-pairs in the obfuscated code are significantly more random than in the unobfuscated code, partly because of disassembly errors caused by “junk bytes” inserted by our obfuscator. The outliers in this figure might serve as starting points for an attacker, but there are dozens of such points, they correspond to thousands of actual opcode pairs in the program, and there is no obvious pattern.

In summary, the individual opcodes and pairs of adjacent opcodes have approximately similar distributions in both unobfuscated and obfuscated programs. Thus, our obfuscation method is on balance quite stealthy.

## 6 Related Work

The earliest work on the topic of binary obfuscation that we are aware of is by Cohen, who proposes overlapping adjacent instructions to fool a disassembler [7]. We are not aware of any actual implementations of this proposal, and our own experiments with this idea proved to be disappointing. More recently, we described an approach to make binaries harder to disassemble using a combination of two techniques: the judicious insertion of “junk bytes” to throw off disassembly; and the use of a device called “branch functions” to make it harder to identify branch targets [20]. These techniques proved effective at thwarting most disassemblers, including the commercial IDA Pro system. Conceptually, this paper can be seen as extending this work by disguising control transfer instructions and inserting misleading control transfers. More recently, we described a way to use signals to disguise the instruction used to make system calls (`int $0x80` in Intel x86 processors), with the goal of preventing injected malware code from finding and executing system calls; this work required kernel modifications. By contrast, the work described in this paper is applicable to arbitrary control transfers in programs and does not require any changes to the kernel. These two differences lead to significant differences between the two approaches in terms of goals, techniques, and effects.

There has been some recent work by Kapoor [14] and Kruegel *et al.* [15] focusing on disassembly techniques aimed specifically at obfuscated binaries. They work around the possibility of “junk bytes” inserted in the instruction stream by producing an *exhaustive disassembly* for each function, i.e., where a recursive disassem-

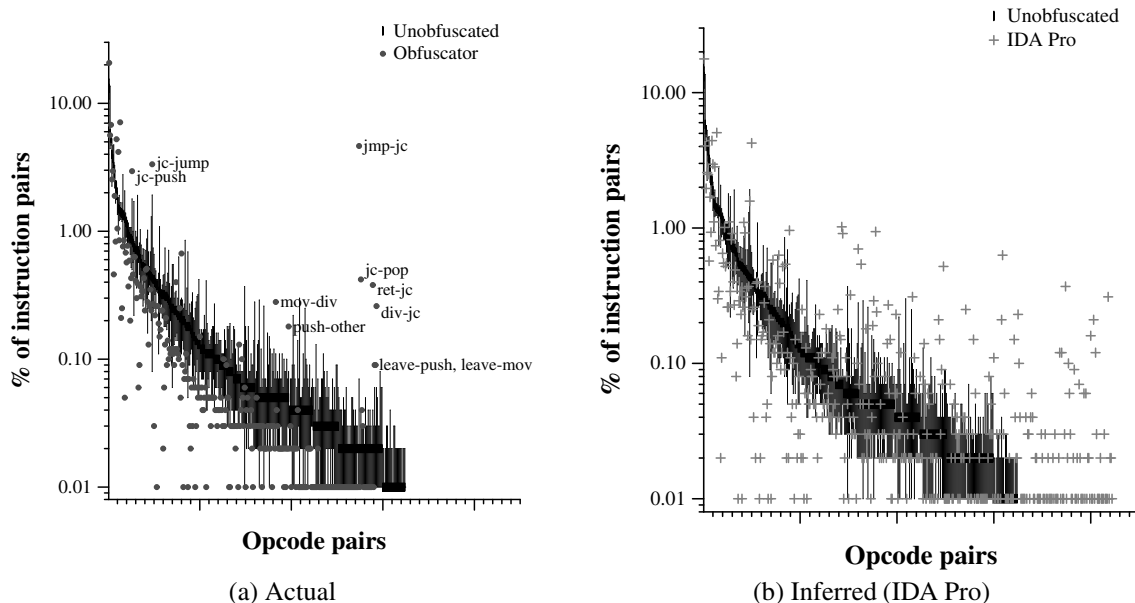


Figure 8: Obfuscation Stealth II: Distribution of Opcode Pairs

bly is produced starting at every byte in the code for that function. This results in a set of alternative disassemblies, not all of which are viable. The disassembler then uses a variety of heuristic and statistical reasoning to rule out alternatives that are unlikely or impossible. To our knowledge, these exhaustive disassemblers are the most sophisticated disassemblers currently available. One of the “attack disassemblers” used for our experiments is an implementation of Kruegel *et al.*’s exhaustive disassembler.

There is a considerable body of work on code obfuscation that focuses on making it harder for an attacker to decompile a program and extract high level semantic information from it [9, 10, 31, 32]. Typically, these authors rely on the use of computationally difficult static analysis problems—e.g., involving complex Boolean expressions, pointers, or indirect control flow—to make it harder to construct a precise control flow graph for a program. Our work is orthogonal to these proposals, and complementary to them. We aim to make a program harder to disassemble correctly, and to thereby sow uncertainty in an attacker’s mind about which portions of a disassembled program have been correctly disassembled and which parts may contain disassembly errors. If the program has already been obfuscated using any of these higher-level obfuscation techniques, our techniques add an additional layer of protection that makes it even harder to decipher the actual structure of the program.

Even greater security may be obtained by maintaining the software in encrypted form and decrypting it as needed during execution, as suggested by Aucsmith [1];

or by using specialized hardware, as discussed by Lie *et al.* [19]. Such approaches have the disadvantages of high performance overhead (in the case of runtime decryption in the absence of specialized hardware support) or a loss of flexibility because the software can no longer be run on stock hardware.

## 7 Conclusions

This paper has described a new approach to obfuscating executable binary programs and evaluated its effectiveness on programs in the SPECint-2000 benchmark suite. Our goals are to make it hard for disassemblers (and humans) to find the real instructions in a binary and to give them a mistaken notion of the actual control flow in the program. To accomplish these goals, we replace many control transfer instructions by traps that cause signals, inject signal handling code that actually effects the original transfers of control, and insert bogus code that further confuses disassemblers. We also use randomization to vary the code we insert so it does not stand out.

These obfuscations confuse even the best disassemblers. On average, the GNU `objdump` program [24] misunderstands over 43% of the original instructions, over-reports the control flow edges by 71%, and misses 63% of the original control flow edges. The IDA Pro system [11], which is considered the best commercial disassembler, fails to disassemble 57% of the original instructions, over-reports control flow edges by 41%, and under-reports control flow edges by 85%. A recent dis-

assembler [15] that has been designed to deal with obfuscated programs fails to disassemble over 55% of the instructions, over-reports control flow edges by 27%, and under-reports control flow edges by over 60%.

These results indicate that we successfully make it hard to disassemble programs, even when we only obfuscate code that is in cold code blocks. If we obfuscate more of the code, we can confuse disassemblers even more. However, our obfuscation method slows down program execution, so there is a tradeoff between the degree of obfuscation and execution time. When we obfuscate only cold code blocks, the average slow-down is 21%, and this result is skewed by three benchmarks for which the training input is not a very good predictor for execution on the reference input. On many programs, the slowdown is negligible. An interesting possibility—which we have not explored but could easily add to our obfuscator—would be selectively to obfuscate some of the hot code, e.g., that which the creator of the code especially wants to conceal.

## Acknowledgements

We are grateful to Christopher Kruegel for the use of the code for his exhaustive disassembler for our experiments.

## References

- [1] D. Aucsmith. Tamper-resistant software: An implementation. In *Information Hiding: First International Workshop: Proceedings*, volume 1174 of *Lecture Notes in Computer Science*, pages 317–333. Springer-Verlag, 1996.
- [2] Black Fenix. Black fenix’s anti-debugging tricks. <http://in.fortunecity.com/y craper/brow er/ / icedete.html>.
- [3] S. Cesare. Linux anti-debugging techniques (fooling the debugger), January 1999. VX Heavens. <http://v .netlu .org/lib/v c .html>.
- [4] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proc. 2005 IEEE Symposium on Security and Privacy (Oakland 2005)*, pages 32–46, May 2005.
- [5] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, Australia, July 1994.
- [6] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software—Practice and Experience*, 25(7):811–829, July 1995.
- [7] F. B. Cohen. Operating system protection through program evolution, 1992. <http://all.net/boo / /evolve.html>.
- [8] R. S. Cohn, D. W. Goodwin, and P. G. Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal*, 9(4):3–20, 1997.
- [9] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. *IEEE Transactions on Software Engineering*, 28(8), August 2002.
- [10] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proc. 1998 IEEE International Conference on Computer Languages*, pages 28–38.
- [11] DataRescue sa/nv, Liège, Belgium. IDA Pro. <http://www.data rescue.com/idaba e/>.
- [12] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with  $O(1)$  worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.
- [13] C. R. Hollander. *Decompilation of object programs*. PhD thesis, Stanford University, 1973.
- [14] A. Kapoor. An approach towards disassembly of malicious binaries. Master’s thesis, University of Louisiana at Lafayette, 2004.
- [15] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proc. 13th USENIX Security Symposium*, August 2004.
- [16] C. Krügel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*, pages 207–226. Springer, 2005.
- [17] E. U. Kumar, A. Kapoor, and A. Lakhotia. DOC – answering the hidden ‘call’ of a virus. *Virus Bulletin*, April 2005.
- [18] A. Lakhotia, E. U. Kumar, and M. Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Transactions on Software Engineering*, 31(11):955–968, 2005.
- [19] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proc. 9th. International Conference on*

- Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.
- [20] C. Linn and S.K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th. ACM Conference on Computer and Communications Security (CCS 2003)*, pages 290–299, October 2003.
  - [21] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. Protecting against unexpected system calls. In *Proc. Usenix Security '05*, pages 239–254, August 2005.
  - [22] R. Muth and S. K. Debray. On the complexity of flow-sensitive dataflow analyses. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL-00)*, pages 67–80, January 2000.
  - [23] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere. `alto`: A link-time optimizer for the Compaq Alpha. *Software—Practice and Experience*, 31:67–101, January 2001.
  - [24] Objdump. *GNU Manuals Online*. GNU Project—Free Software Foundation. [www.gnu.org/manual/binutil-2.14/html.chapter/binutil-2.14.html](http://www.gnu.org/manual/binutil-2.14/html.chapter/binutil-2.14.html).
  - [25] M. Prasad and T. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proc. USENIX Technical Conference*, June 2003.
  - [26] B. Schwarz, S. K. Debray, and G. R. Andrews. Disassembly of executable code revisited. In *Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE)*, pages 45–54, October 2002.
  - [27] P. K. Singh, M. Mohammed, and A. Lakhotia. Using static analysis and verification for analyzing virus and worm programs. In *Proc. 2nd. European Conference on Information Warfare*, June 2003.
  - [28] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.
  - [29] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.
  - [30] H. Theiling. Extracting safe and precise control flow from binaries. In *Proc. 7th Conference on Real-Time Computing Systems and Applications*, December 2000.
  - [31] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Proc. International Conference of Dependable Systems and Networks*, July 2001.
  - [32] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, Dept. of Computer Science, University of Virginia, 12 2000.
  - [33] Z. Xu, B. P. Miller, and T. Reps. Safety checking of machine code. In *Proc. ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 70–82, June 18–21, 2000.

## Appendix — Randomizing the Computation of Values

The essential idea is to carry out multiple, random, value-preserving rewritings of the syntax tree for an expression. We start with a simple expression, e.g., an integer constant or a variable, and repeatedly rewrite it, using value-preserving transformation rules, to produce an equivalent expression (i.e., one that will always evaluate to the same value).

Figure 9 gives a non-exhaustive list of example rewrite rules. In the rules, we use ‘ $x$ ’ and ‘ $y$ ’ to denote variables, i.e., the values of registers or memory locations; ‘ $k$ ’, ‘ $m$ ’, and ‘ $n$ ’ to denote integer constants; and  $a$  to denote something that is either a variable or a constant. Note that equivalences that hold for integers may not hold at the machine level, e.g.,  $(x + 1) - 1$  need not evaluate to  $x$ . Thus, in general we cannot use associative and distributive laws for rewriting.

The expression being rewritten is maintained as a syntax tree. Initially, the tree consists of a single node, namely, the variable or constant being rewritten. Each node of the tree has an associated label indicating what kind of value is being computed (zero, nonzero, arbitrary, etc.). The rewriting proceeds as follows. We first choose a positive random value as the number of rewriting steps. Each rewriting step consists of the following:

1. Randomly choose a leaf node  $X$  of the tree.
2. Randomly choose a rewrite rule  $R \equiv Y \longrightarrow E$  from the set of rules corresponding to the label of the chosen leaf node.
3. Modify the syntax tree by adding the appropriate instance of  $E$  (i.e., with all occurrences of  $Y$  replaced by  $X$ ) as child nodes of  $X$  and update the set of leaf nodes appropriately.

<u>Zero:</u>	$0 \longrightarrow 0 + 0$	
	$0 \longrightarrow a \ a$	
	$0 \longrightarrow 0 \ a$	
	$0 \longrightarrow k \ 0 : a$	$k \neq 0$ ; arbitrary $a$
<u>Nonzero</u> $k$ :	$k \longrightarrow 0$	
	$k \longrightarrow m \{ \text{rotr}, \text{rotr} \} n$	$m \neq 0$ , and any $n$ ( $\text{rotr} = \text{rotate right}$ ; $\text{rotr} = \text{rotate left}$ )
	$k \longrightarrow k \ n$	$n = w - m$ , where $m$ is the position of the least significant '1' bit in $k$ , and $w$ is the machine word size in bits.
	$k \longrightarrow m \ n$	$m \neq n$
<u>Arbitrary</u> $x$ :	$x \longrightarrow 0 \ x$	
	$x \longrightarrow x \ 1$	
	$x \longrightarrow 0 \ y : x$	$y$ is any value
	$x \longrightarrow m \ x : y$	$y$ is any value; $m \neq 0$

Figure 9: A (non-exhaustive) list of rewriting rules (Operators are as in the C language)

The rewritten expression may contain “free variables,” i.e., variables that are not initialized to any value. The value of the overall expression does not depend on the actual value taken on by such a free variable, so any value will do. In our implementation, we simply use the contents of any arbitrary register or legal memory location for such variables.

Once the rewritten expression has been generated, we generate code for it via a straightforward post-order traversal of the final syntax tree.

# Active Hardware Metering for Intellectual Property Protection and Security

Yousra M. Alkabani  
Computer Science Dept.  
Rice University, Houston, TX  
yousra@rice.edu

Farinaz Koushanfar  
Electrical and Computer Engineering Dept.  
Rice University, Houston, TX  
farinaz@rice.edu

## Abstract

We introduce the first *active hardware metering* scheme that aims to protect integrated circuits (IC) intellectual property (IP) against piracy and runtime tampering. The novel metering method simultaneously employs *inherent unclonable variability* in modern manufacturing technology, and *functionality preserving alternations* of the structural IC specifications. Active metering works by enabling the designers to lock each IC and to remotely disable it. The objectives are realized by adding new states and transitions to the original finite state machine (FSM) to create boosted finite state machines (BFSM) of the pertinent design. A unique and unpredictable ID generated by an IC is utilized to place an BFSM into the *power-up state* upon activation. The designer, knowing the transition table, is the only one who can generate input sequences required to bring the BFSM into the *functional initial (reset) state*. To facilitate remote disabling of ICs, *black hole* states are integrated within the BFSM.

We introduce nine types of *potential attacks* against the proposed active metering method. We further describe a number of countermeasures that must be taken to preserve the security of active metering against the potential attacks. The implementation details of the method with the objectives of being low-overhead, unclonable, obfuscated, stable, while having a diverse set of keys is presented. The active metering method was implemented, synthesized and mapped on the standard benchmark circuits. Experimental evaluations illustrate that the method has a low-overhead in terms of power, delay, and area, while it is extremely resilient against the considered attacks.

## 1 Introduction

In the dominant horizontal semiconductor business model, piracy (illegal copying) and tampering of hardware are omnipresent. In the horizontal business model,

hardware IP<sup>1</sup> designed by the leading edge designers are mostly manufactured in untrusted offshore countries with lower labor and operational cost. This places the designers in an unusual *asymmetric* relationship: the designed IP is transparent to the manufacturers, but the fabrication process, quantity and added circuitry to the manufactured integrated circuits (ICs) by the foundry are clandestine to the designers and IP providers.

The security threat, financial loss and economic impacts of hardware piracy which have received far less attention compared to software, is even more dramatic than software [8, 31]. Software piracy has received more attention compared to hardware also because it requires low-cost resources that are available to the general public. Protection of hardware is also crucially important because the ICs are pervasively used in almost all electronic devices and the potentially adversarial fabrication house has the full control over the hardware resources being manufactured. It is estimated that the computer hardware, computer peripherals, and embedded systems are the dominant pirated IP components [31].

Several other issues make the IC protection problems truly challenging: (i) very little is known about the current and potential IC tampering attacks; (ii) numerous attacking strategies exist, since tampering can be conducted at many levels of abstraction of the synthesis process; (iii) the most likely hardware adversaries are financially strong foundries and foreign governments with large economic resources and technological expertise; (iv) the adversary has full access to the structural specification of the design and most often also to the manufacturing test vectors; (v) the internal part of manufactured ICs are intrinsically opaque. While it is possible to tomographically scan an IC, the dense metal interconnect in 8 or more layers of modern manufacturing technology greatly reduce the effectiveness of such expensive inspections.

IC *metering* is a set of security protocols that enable the design house to gain post-fabrication control by pas-

sive or active count of the produced ICs, their properties and use, or by remote runtime disabling.

Our strategic goal is the *development, implementation, and quantitative evaluation of symmetric mechanisms and protocols for hardware protection procured by untrusted synthesis, manufacturing, and/or testing facilities*. The term *symmetric* emphasizes that both the designers and the foundry will be protected by the new methods. The symmetry is warranted by the unique variabilities and the key exchange mechanism that is based on the agreement of both parties for unlocking each IC.

Hardware metering is important from both commercial and military point of views. For example, without metering, a foundry can produce numerous copies of one design without paying royalties, or, as another example, the sensitive defense designs may become available to adversaries. The *passive hardware metering* schemes work by giving a unique ID to each chip [17, 20, 21]. The *first ever active hardware metering* method introduced in this paper, provides not just mechanisms for detection of illegal copies, but more importantly, ensures that no manufactured IC can be used without the explicit consent of the designer.

The proposed methods employ two generic security mechanisms: (1) *uniqueness of each IC due to manufacturing variability*; and (2) *structural manipulation of the design specification while preserving behavioral specification*. While the first mechanism has been already proposed and used for unique IC identification, the second is novel. Even more novel is the integration of two mechanisms, a task that requires a great deal of creativity and formation of solutions to a spectrum of challenging technology, synthesis and optimization problems, with a greater impact than the sum of the powers of the individual techniques.

The integration to the functionality is performed by interwinding the unique unclonable IDs for each chip into the FSM of the design. The integrated control part is denoted by BFSM, and is built by adding new states and transitions to the original FSM, while preserving the original functionality of the circuit. To bring the BFSM into the functional initial (reset) state, knowledge of the transition table is required. Since the designer is the only one who knows this information, no one else can generate a key with a finite amount of resources to unlock the IC. Using a combination of BFSM and newly added black hole states, remote disabling of the ICs can be made possible. We outline several possible attacks against the introduced active hardware metering method and provide mechanisms that neutralize the impact of those attacks. For example, we show how addition of the black hole states disable the random guessing attacks.

The remainder of the paper is as follows. After describing the background, flow and the state-of-the-art in

the next two sections, we represent the active metering method in Section 4. In Section 5, we show a low-overhead implementation and obfuscation of active metering. Section 6 introduces potential attacks and the countermeasures that needs to be taken to be resilient against the attacks. We present experimental evaluation of the prototype implementation on several standard design benchmarks in Section 7. We outline a number of potential applications in Section 8 and conclude in Section 9.

## 2 Preliminaries

In this section, we describe the necessary background required for understanding the active hardware metering approach. The aim is to make the paper self-contained for the readers who are not familiar with the hardware design and synthesis process. Next, we describe the global flow of the active hardware metering approach.

### 2.1 Background

**Manufacturing variability (MV).** The intense industrial miniaturization of CMOS devices has been driven by the quest for increasing computational speed and device density, while lowering cost-per-function, as predicted by Moore's law. CMOS variations result in high variability in the delay and the currents of the VLSI circuits. The variations might be temporal or spatial. The temporal variations may occur across nanoseconds to years [24]. Spatial variation is due to lateral and vertical differences from intended polygon dimensions and film thicknesses. Spatial variation may be intra-die, or inter-die [27]. Aside from device variations, the circuit response and its variability are correlated with circuit topology. We will utilize the spatial variations in our benefit, while we address the problem of alleviating temporal variability. Bernstein et al. provide a classification of device variations (beyond 65nm) [4].

**Design descriptions.** We consider the case in which the sequential design in question represents a fully synchronous flow and that the description of its functionality from an input/output (I/O) perspective is publicly available. We assume that the functionality is fully fixed, in that the I/O behavior is fully specified. Therefore, we utilize unique unclonable identification to embed a distinct mark in the functionality of each IC, without altering the functionality in terms of the normal I/O behavior of the circuit. Our technique is applicable to the case where the piece of IP is available in structural HDL description, or in form of a netlist that may or may not be technology dependent. The description uniquely defines the sequential circuit's behavior and the state transition graph (see

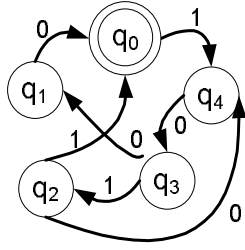


Figure 1: Example of a STG with five states. The inputs required for state-to-state transition are shown next to the edges.

the next subsection) of the design.

During the design flow, the user will take such a description and if required, will map it to a specific technology. Typically, logic level optimizations such as retiming are performed at this stage. Most often, the circuit is used as a part of a more complex design.

**Finite state machine (FSM).** FSM is a discrete dynamical structure that translates sequences of input vectors into sequences of output vectors. FSM can represent any regular sequential function. It appears in different forms, e.g. case statements in VHDL and Verilog HDL. The FSM is defined by the tuple  $M=(\Sigma, \Delta, Q, q_0, \delta, \lambda)$ , where  $\Sigma \neq \emptyset$  and  $\Delta \neq \emptyset$  are a finite set of inputs and outputs symbols respectively;  $Q=\{q_0, q_1, \dots\} \neq \emptyset$  is a finite set of states while  $q_0$  is the "reset" state; and the transition function is denoted as  $\delta(q, a)$  on the input  $a$  and the set  $Q \times \Sigma \rightarrow Q$ , while the output function is denoted as  $\lambda(q, a)$  on the set  $Q \times \Sigma \rightarrow \Delta$ .

To represent the state transitions and output functions of the FSM, we use the state transition graph (STG), with nodes corresponding to states and edges defining the input/output conditions for a state-to-state transition. An example STG is shown in Figure 1, where there are five states  $\{q_0, q_1, q_2, q_3, q_4\}$ ,  $q_0$  is the reset state, and there is a one-bit input controlling the state-to-state transitions. In the remainder of the paper, we use the terms STG and FSM interchangeably to refer to the control part of the design.

## 2.2 Global flow

As a motivational example for our problem, consider the scenario in which a given hardware intellectual property (IP) that belongs to its legitimate owner (Alice) is made available to a fabrication house (Bob). Alice pays for and demands  $N_A$  ICs implementing its design. Bob, utilizes the IP description to construct a mask that implements the design. Bob employs the mask to make  $N_A + N_B$  copies of the design, where the illegal  $N_B$  copies do not encounter much additional cost due to the availability of

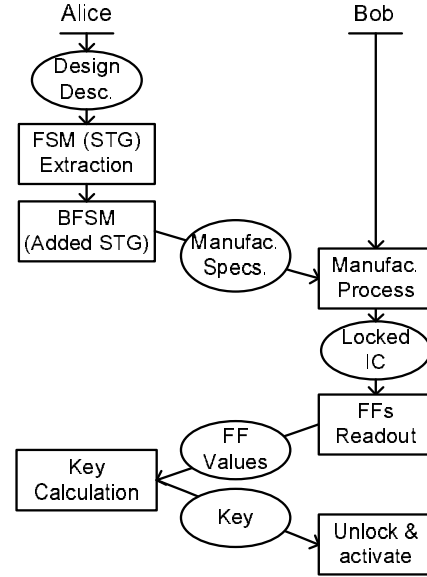


Figure 2: The global flow of the active hardware metering approach.

the mask. Bob may sell the  $N_B$  illegal copies and make a lot of profit with negligible additional overhead.

The novel active metering helps Alice to protect her design against piracy by manipulating the STG of the original design, with the objective of creating a locked state, that is unique for each of the ICs manufactured from the design with a very high probability. Upon manufacturing by Bob, each device will be uniquely locked (i.e., rendered non-functional), unless Alice is contacted by Bob to provide the particular key to unlock the IC. The scheme gives the full control over the manufactured parts and operational devices from the IP to Alice.

The global flow of the active hardware metering method is shown in Figure 2. We now describe the figure step by step. Alice takes the high level design description and synthesizes it to get the FSM of the design. Next, she constructs the BFSM by adding extra states. After that, she sends the detailed manufacturable design specifications to Bob who makes the mask and manufactures multiple ICs implementing the design. The manufactured ICs are locked (nonfunctional) at this stage. For each IC, Bob reads out the values in its flip flops (FFs) and sends the values to Alice. FF values can be read nondestructively, and the values are unique for each IC. Alice, knowing the BFSM structure, computes a specific key that can be used as input to that IC for unlocking it. The key is then sent back to Bob who utilizes it to activate the IC.

### 3 Related work

We survey the related literature that has influenced and inspired this work along four main lines of research: variability-based ID generation, authentication and security by variability-based IDs, intellectual property protection of VLSI designs, and invasive and noninvasive hardware attacks.

A number of authors have proposed and implemented the idea of addition of circuitry that exploits manufacturing variability to generate unique random sequence (ID) for each chip with the same mask [20, 21, 28]. The IDs are unclonable and separated from the functionality and do not provide a measure of trust, as they are easy to tamper and remove. Loftstrom et al. proposed a method for mismatching the devices based on changing the threshold of the circuits by placing the impurity of random dopant atoms [20]. Maeda et al. proposed implementing the random IDs on poly-crystalline silicon thin film transistors [21]. The drawback of the two described approaches is that they both need specialized process technology, and are easily detectable. Very recently, Su et al. have proposed a technique to generate random IDs by using the threshold mismatches of two NOR gates that are positively feeding back each other [28]. We will exploit their technique for the random ID generation.

A team of researchers has explored the idea of using variability-induced delays for authentication and security [9, 19, 29]. They use Physically Unclonable Functions (PUFs) that map a set of challenges to a set of responses, based on an intractably complex physical system. PUFs are unique, since process variations cause significant delay differences among ICs coming from the same mask. For each IC, a database of challenge-response sets is needed. Authentication occurs when the IC correctly finds the output of one or more challenge inputs. PUF-based methods solely utilize manufacturing variability as their security mechanism. *In contrast, our proposed methods introduce a paradigm shift in hardware security by adding new strong mechanisms: integration into circuit functionality at the behavioral synthesis level. Furthermore, even though the active metering methods can be utilized for authentication, its main target is addressing the hardware piracy problem.*

Koushanfar et al. have introduced the first hardware metering scheme that gives unique IDs to each IC [16]. The scheme was to make a small part of the design programmable so that one could upload different control paths post fabrication. They further described how to generate numerous different instances of the same control path with the same hardware [17]. They have also provided probabilistic proofs for the number of identical copies and probability of fraud for the proposed metering schemes [16, 17]. All metering schemes were pas-

sive. Indeed, no active metering scheme has been proposed to date. The prior work in trusted IC domain also includes introduction of several *watermarking* schemes that integrate watermarks to the functionality of the design at the behavioral synthesis level [11–13, 15, 22, 23, 30, 32]. *Watermarking is a fundamentally different problem when compared to metering. It addresses the problem of uniquely identifying each IP and not identifying each IC, so the existence of the same mask does not affect the watermarking results.* Fingerprinting for unique identification of programmable platforms has been proposed [18], but the techniques are not applicable to application specific designs (ASICs) due to the existence of a unique mask. Qu and Potkonjak provide a comprehensive survey of the watermarking, fingerprinting and other hardware intellectual property protection methods [23].

Even though many strong cryptographical techniques are available in hardware and software, their attack resiliency has been only verified by classical cryptanalysis methods. A class of attacks that is very challenging to address consists of physical techniques. Physical attacks take advantage of implementation-specific characteristics of cryptographical devices to recover the secret parameters. Koeune and Standaert provide a tutorial on physical security and side-channel effects [14]. The physical attacks are divided into invasive and non-invasive [3]. Invasive attacks depackage the chip to get direct access to its inside, e.g., probing. Noninvasive attacks rely on outside measurements, e.g., from the pins or by X-raying the chip, without physically tampering it.

There are multiple ways to attack an IC, including probing, fault injection, timing, power analysis, and electromagnetic analysis. Invasive attacks are typically more expensive than the noninvasive ones, since they need individual probing of each IC. Note that, according to the well-established taxonomy of physical attacks, attacks by the funded organizations (e.g., foundries) are the most severe ones, since they have both the funding and technology resources [1–3].

### 4 Active hardware metering

In this section, we present the details of the active hardware metering approach. Active metering is *integrated into the standard synthesis flow*, and is *low overhead*, *generalizable*, and *resilient against attacks*. By generalizable, we mean that the lock can be implemented on structures that are common to all designs. By attack-resiliency, we mean the cryptographic notion of a lock: that an attacker that does not have infinite computational power should not be able to unlock the IC without the knowledge of a key. To be generalizable, the method proposed here aims at protecting the design by boosting the design's FSM (and creating a BFSM) common to

the widely used class of sequential designs. In this section, we describe the BFSM construction and introduce the locking mechanism. Implementation details are discussed in the next section.

## 4.1 Method

**Random Unique Block (RUB).** Perhaps the most important component of the proposed security mechanism is the existence of the unclonable unique ID for each IC. The IDs are a function of the variability present at each chip and are therefore, specific to the chip. RUB is a small circuitry added to the design, whose function is to generate the unique ID. It is desirable that the RUBs do not change and remain stable over time. Recently, a few paradigms for designing unique identification circuitry was proposed [20, 21, 28]. The resulting IDs are mostly stable, and we will later show how to extract a nonvolatile ID from the RUB, even in presence of a few unstable bits.

**Addition of the BFSM.** The key idea underlying the proposed active metering scheme can be described in a simple way. Assume that the original design contains  $m$  distinct states. Further assume that the state of STG are stored in  $k$ , 1-bit flip flops (FFs). The FFs represent a total of  $2^k$  states, out of which  $m$  states correspond to the original design and  $(2^k - m)$  states are don't cares. The metering mechanism adds an extra part to the FSM of the design. The added states are devised such that there are a number of transitions from the states in the added STG to the reset state  $q_0$  of the original design.

In our scheme, the power-up state of each IC is built to be a function of the manufacturing variability and thus, will be unique to each instance. Furthermore, we select  $k$  such that  $2^k - m \gg m$ . This selection ensures that when the circuit is powered up, its initial state will be in one of the added states in BFSM. Assume that the IC is powered up in the added state  $q_{a0}$ . During the standard testing phase, the manufacturer can read the state of the design, e.g., by scanning and reading the FF's. However, unless the foundry has the knowledge of the STG, finding the sequence of inputs required for the correct transition from the state  $q_{a0}$  to the reset state  $q_0$  is a problem of exponential complexity. Essentially, there will be no way of finding the sequence other than trying all the possible combinations.

More formally, assume that the sequence of  $I$  primary inputs denoted as  $\alpha_I = \{a_1, a_2, \dots, a_I\}$  applied to the state  $q_{a0}$  is one correct sequence of states that starts from  $q_{a0}$  traverses  $I$  states denoted by  $Q_I = \{q_{a1}, q_{a2}, \dots, q_{a(I-1)}, q_0\}$ , i.e.,  $q_0 = \delta(q_{a0}, \alpha)$ . Assuming that the input is  $b$  bits and there are cycles in STG, finding the correct input sequence that would result in  $I$  consecutive correct transitions is a problem with exponential complexity

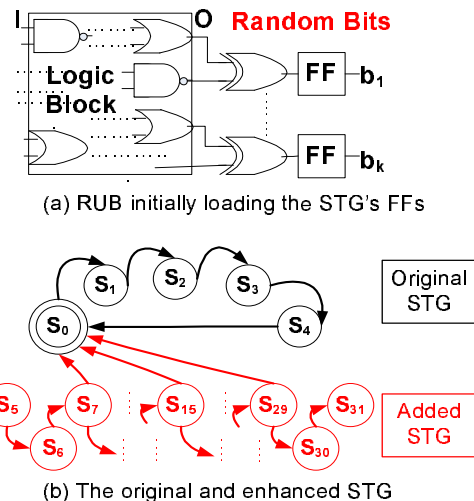


Figure 3: The boosted FSM (BFSM).

with respect to  $b$  and is thus, intractable.

As an example, consider the STG shown in Figure 3(b) that consists of the original STG that has five states ( $\{q_0, q_1, q_2, q_3, q_4\}$ ) with augmentation of twenty seven added states ( $\{q_5, q_6, \dots, q_{31}\}$ ). Edges are incorporated to the added states to ensure that there are paths from each of the added states to the reset state of the design. The block shown in Figure 3(a) is a RUB.

The output of the RUB defines  $k$  random bits that will be loaded into the FF's of the augmented STG upon start-up. Now, an uninformed user who does not have the information about the transition table (e.g., foundry), can readout the data about the initial added state  $q_{a0}$ , but this information is not sufficient for finding the sequence of primary input combinations to arrive at the reset state  $q_0$ . However, the person who has the information about the structure of the STG, upon receiving the correct state, would exactly know how to traverse from this locked state to  $q_0$ . In other words, the owner of the FSM description is the only entity who would have the key to unlock the IC.

An interesting application of the proposed BFSM construction method is in remote disabling. Alice will save the RUBs and the keys for all the ICs that she has unlocked. Using the chip IDs that are integrated within the functionality, she can add mechanisms that enable her to monitor the activities of the registered chips remotely, for example, if they are connected to the Internet. She can further add transitions from the original STG to untraversed states, to lock the IC in case it is needed. Remote disabling has a lot of applications. For example, it can be used for selective remote programming of the devices, and royalty enforcement.

## 4.2 Ensuring proper operation

The following issues and observations ensure proper operation and low-overhead of active hw metering:

(i) **Storing the input sequence (key) for traversal to the initial state  $q_0$ .** During testing, once Bob scans out the FF values and sends them to Alice, she provides the key to Bob. He includes both the original RUB and the key in the chip, for example, in a nonvolatile memory. This data is utilized along with the unclonable RUB circuit, for transition to the reset state. Since the power-up state is unique for each IC, sequence of inputs (key) that traverse the power-up state to the reset state is also specific to each IC. One needs to store the key which performs the traversal at the power-up state on each chip. There are many ways to accomplish this. For example, the designer could add a small programmable part to the design which needs to be *coded* with the unique sequence (key) before each IC is in operation. Coding ensures protection of keys against other software attacks. As an alternative, the sequence might not be included in the memory and just used as a permanent password to the IC.

(ii) **Powering up in one of the added states.** This condition can be easily guaranteed by selecting a large enough  $k$ . Assuming that all the states have an equal probability, the probability of starting in one of the added states is  $(2^k - m)/2^k$ . For a given  $m$ , we select  $k$  such that the probability of not being in one of the added states is smaller than a given probability. For example, for  $m = 100$  and  $k = 30$ , the probability of starting up in an original state is less than  $10^{-7}$ .

(iii) **Diversity of power-up states (unique IDs).**  $k$  should also be selected so that the probability of two ICs having the same ID becomes very low. Assume that we need to have  $d$  distinct ICs each with a unique ID. Assuming that the IDs are completely random and independent, we utilize the Birthday paradox to calculate this probability and to make it low. Consider the probability  $P_{ICID}(k, d)$  that no two ICs out of a group of  $d$  will have matching IDs out of  $2^k$  equally possible IDs. Start with an arbitrary chip's ID. The probability that the second chip's ID is different is  $(2^k - 1)/2^k$ . Similarly, the probability that the third IC's ID is different from the first two is  $[(2^k - 1)/2^k] \cdot [(2^k - 2)/2^k]$ . The same computation can be extended through the  $2^k$ -th ID. More formally,

$$\begin{aligned} P_{ICID}(k, d) &= \frac{2^k - 1}{2^k} \cdot \frac{2^k - 2}{2^k} \cdot \dots \cdot \frac{2^k - (d - 1)}{2^k} \\ &= \frac{2^k!}{(2^k - d)! 2^{dk}} \end{aligned} \quad (1)$$

Thus, knowing  $d$ , the number of required distinct copies,

and setting a low value for  $P_{ICID}$ , we would be able to find  $k$  that satisfies the above equation.

(iv) **Overhead of the added STG.** The number of states increases exponentially with adding each new bit, and thus, the scheme has a very low overhead. Note that, in modern designs, the control path of the design (i.e., FSM) is less than 1% of the total area and hence, adding a small overhead to the FSM does not significantly affect the total area [7, 10]. In the next section, we will describe a low-overhead implementation of the proposed method.

(v) **Diversity of keys.** There is a need to ensure that the keys are distinct in all parts of their sequences, or there is a very small shared subsequence between different keys. This is granted by making multiple paths on the graph from each of the states to the reset state. We will elaborate more on this issue in the attack resiliency section.

## 5 Low overhead implementation and obfuscation

In this section, we discuss the implementation details of the RUB and the BFSM that are the required building blocks for the active hardware metering approach. We start by outlining the desired properties of each block, and then we delve into its implementation details.

### 5.1 RUB implementation

A critical aspect of the proposed security and protection mechanisms is the generation of random ID bits. There are a number of properties that the RUB implementation has to satisfy, including:

- **Low overhead.** The added parts must not introduce a significant additional overhead in terms of delay, power consumption and the area.
- **Distribution of IDs and their correlations.** To have the maximal difference between any two ID numbers (the maximal Hamming distances) the ID bits must be completely random. Thus, no correlation must be present among the ID bits on the same die or across various dies.
- **Indiscernibility.** The IDs must be integrated within the design, such that they cannot be discerned by studying the layout of the circuit. For example, the IDs should not be placed in a memory-like array, where the regularity of the array and its connections to the FFs could be easily detected.
- **Stability.** There is a need to stabilize the IDs over the lifetime of an IC. This is particularly important since studies have shown the temporal changes in CMOS process variations due to many environmental and aging ef-

fects including, residual charges, self-heating, negative-bias temperature instability, and hot electron effects [4].

For implementing the random IDs, we employed the recent novel approach proposed by Su et al. [28]. They have designed and tested a new CMOS random ID generation circuit that relies on digital latch threshold offset voltages. Using cross coupling of gates, they report significant improvement in readout speed and power consumption over the existing designs.

Each ID bit is generated by cross-coupled NOR gates. The latch sides are pulled low initially. At the high to low clock transition, the state of each latch is determined by the threshold voltage mismatch of the transistors. Essentially, the approach relies on the positive feedback inherent in the latch configuration to amplify the mismatch. This design removes the need for comparators, low offset amplifiers, or extra dopants needed in previous random ID generation methods [9, 20]. The nominal overhead of the above proposed approach is two NOR gates per bit. The authors have reported 96% stable IDs using this method, while using dummy latches to protect the IDs.

Even though we use the random bit architecture described above, our layout and implementation of random bits are very different. To be indiscernible, we do not place the coupled NOR gates in an array, and instead synthesize them with the rest of the circuit and camouflage them within the sea of gates. based on invariability of the ID bits for an IC. In Subsection 6.2, we provide a mechanism that ensures the occasional errors in ID bits do not affect the hardware metering approach.

## 5.2 BFSM implementation

The key design objectives and challenges of the BFSM are as follows:

- **Low overhead.** The addition of the states to the original FSM must have a low overhead in terms of area, power, and delay. This is particularly challenging: as we have computed in Subsection 4.2, even under the assumption of having RUBs with Uniform distribution of random bits, the number of added states must be exponentially high to ensure a proper operation.
- **Traversal path.** There must be a path on the BFSM, from each of the power-up states (except for the black hole states that we will describe in Section 6.2) to the reset state.
- **States obfuscation.** The states must be completely obfuscated and interchanged to camouflage the added STG and the original STG. Another level of obfuscation is disabling the observability of the FFs, so that similar states on two ICs do not exactly have the same code scanned out from their FFs.

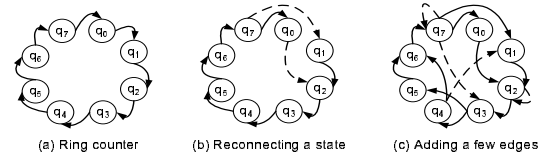


Figure 4: Illustration of steps for building a sparse 3-bit STG.

• **Multiplicity of keys.** It is highly desirable to construct the paths on the added STG in such a way that there are multiple paths from each power-up state to the reset state. This will ensure that there are multiple keys for traversal. Now, if the states are obfuscated such that a similar state on two ICs has different codes, and each of them gets a different key for traversal to the original STG, the state similarities will not be apparent, even to a smart observer.

To achieve a low overhead, we have systematically designed STG blocks that are capable of producing an exponential number of states with respect to their underlying hardware resources. The blocks are designed such that there are multiple paths from each of the added states to the reset state and thus, the multiplicity of the keys is satisfied. Our first attempt was to synthesize the added blocks of STG and the original STG together. However, because the synthesis software automatically optimizes the interwoven architecture, it most often ended up with a combined STG that was much larger than the sum of its components. Thus, we decided to first separately synthesize the original and the added STG before we merge them. Next, we employed obfuscation methods that constantly alter the values of the FFs, even those that are not used in state assignment in the current STG. As we will see in attack resiliency section, the introduced obfuscation method has the side-benefit that the adversary cannot exactly distinguish a similar state on two different ICs.

The added STG can be designed to be low overhead; there are exponentially many states for each added FF, ignoring the overhead of the STG edges. However, in real situations, the transitions (edges) require logic. Thus, the added STG is constrained to be sparse to satisfy a low overhead. We have built this block in a modular way. We describe one of our modules here and then discuss systematically interconnecting the modules to have a multi-bit added STG that has a low overhead.

The first module is a 3-bit added STG. In Figure 4, we show three steps for building this module. We start by a ring counter as shown in Graph 4(a). Next, we pick a few states and reconnect them to break the regularity. A small example is illustrated in Graph 4(b), where the state  $q_1$  is reconnected, such that still there will be a path from each state to any other state. Finally, we add a few transitions (edges) to the STG, like the example shown

in Graph 4(c); here the states  $q_1$  and  $q_4$  are reconnected, while the edges  $\{q_4 \rightarrow q_1, q_7 \rightarrow q_3, q_7 \rightarrow q_7, q_2 \rightarrow q_2\}$  are added.

The example is just an illustration. Many other configurations are possible. The various combinations have different post-synthesis overhead. To ensure a low-overhead, we exhaustively searched the synthesized 3-bit structure with various sparse edge configurations like the example above, and selected the configurations with the lowest overhead as our 3-bit modules. As it is apparent from the structure, many low-overhead configurations are possible and we do not need to use the same module multiple times.

After that, we picked the low overhead modules and started to add edges to interconnect them, such that the connectivity property is satisfied, and the interconnected configuration still has low overhead. Furthermore, we need multiple interconnecting paths that can produce multiple keys. This is again done via a modular randomized edge addition and searching the space of the synthesized circuits to find the best multi-bit configurations. Note that, the synthesis program performs state-encoding for the interconnected modules. We have noticed that the distance of the codes assigned to the states does not have a correlation with the proximity of the states. Therefore, even for two RUBs that are only different in 1-bit, typically the power-up states are not close-by on the added STG.

In our experiment, we have tested our approach on 12, 15, and 18-bit added STGs. Now, the original STG has to be glued to the added part. This is done by an obfuscation scheme that ensures the states of FFs that are associated with the original STG keep pseudorandomly changing, even when we traverse the states of the added STGs. Thus, for an observer who studies the values of the interleaved FFs, the activity study would not yield an informative conclusion that can help separating the original and the added states. A simple example for this obfuscation is depicted in Figure 5. In this figure, a small original STG with five states is presented. The cloud shown below the original STG indicates the added states.

There are multiple state transitions from the added states to the original state. However, we only show one arrow on the plot not to make it more crowded. In this example, we use the three don't cares of the design for obfuscation purposes. There are 3 don't care states that we use to form three new dummy states  $q^*_5$ ,  $q^*_6$ , and  $q^*_7$ , illustrated in grey color. The glue logic attaches the inputs and the states of the added STG to the dummy STG. Thus, by carefully designing, one can alter the bits on the dummy STG by changing the input and the states of the added STG without touching the original FSM. If the design does not have sufficient don't cares, we can add a couple of FFs for the dummy states and use the same

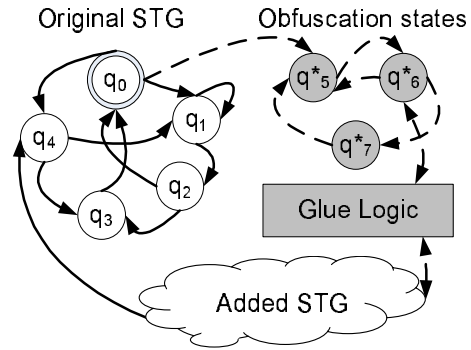


Figure 5: Obfuscation of the original STG.

paradigm. The important requirement for the dummy states is that as a group they should present both 1 and 0 digits in all FFs. The original STG is also connected to the dummy STG and can utilize it as a black hole (described more thoroughly in Subsection 6.2), if there is a need to halt the IC.

## 6 Attack resiliency

This section first identifies several types of potential attacks on the active hardware metering approach. Next, we outline a number of mechanisms that must be added to the basic active metering scheme to ensure its resiliency against the suggested attacks.

The adversary (Bob) may attempt to perform a set of invasive or noninvasive attacks on the proposed active metering scheme. Bob may do so by measuring and probing one instance, or by statistically studying a collection of instances. In this section, we first identify and describe the attacks. Next, we propose efficient countermeasures that can be taken to neutralize the effect of potential adversarial acts.

We assume that Bob knows all the concepts of the proposed hardware metering scheme, has the complete knowledge of the design at all levels of abstraction provided to the foundry (e.g., logic synthesis level netlist, and physical design GDS-II file, but no behavioral specification), can simultaneously observe all signals (data) on all interconnects and flip-flops (FFs), and can measure, with no error, all timing characteristics of all gates in the ICs.

### 6.1 Description of attacks

The starting point for development and evaluation of the metering schemes is identification and specification of several types of potential attacks:

(i) **Brute-force attack.** Bob aims to place the pertinent IC into the initial state by systematically applying the

input sequences to the BFSM. The systematic application may be a randomized strategy, or may be based on scanning the FFs. Brute-force attack works by randomly changing the inputs in hope of arriving at the reset state. Scanning works by reading out the FF values for a few ICs and storing them. The FFs in the current IC are then monitored for the existence of a common state with the stored ones. In case a state that was read in the previous ICs is reached, Bob uses the same key for traversal to the reset state.

**(ii) Reverse engineering of FSM.** Bob may try to scan the FFs to extract the STG. The attempt would be to remove the added STG from the BFSM, to separate the original and the added states.

**(iii) Combinational redundancy removal.** Bob may use the combinational redundancy removal, a procedure that attempts to remove the combinational logic that is not necessary for the correct behavior of the circuit. The proposed techniques of this class often take into account the set of reachable states of the FSM under examination [25]. Note that, the attacks that were described so far can greatly benefit from the ability to simultaneously monitor the multitude of signals/values on the IC using laser reading.

**(iv) RUB emulation.** The goal of this attack is to create a reconfigurable implementation capable of realizing hardware that has the identical functional and timing characteristics to a RUB for which a legal key is already received.

**(v) Initial power-up state capturing and replaying (CAR).** Bob knows the initial power-up state of an unlocked IC. He can use invasive methods to load the FFs of other ICs to the same power-up state as the unlocked IC and then utilize the same key to decode the new locks. Note that, unless invasive methods are used, the only way for Bob to alter the values in the FFs is to change the states using the input pins. Without the knowledge of the STG, the change of state can only be done as described in the first attack. This attack and the next two belong to the class of replay attacks.

**(vi) Initial reset state CAR.** Bob scans the FF of an unlocked IC and reads the code of the reset state. Next, he employs invasive methods to load the FFs of other ICs to unlock them.

**(vii) Control signals CAR.** In this attack, Bob attempts to bypass the FSM by learning the control signals and attempting to emulate them. Bob may completely bypass FSM by creating a new FSM that provides control signals to all functional units, and control logic (e.g. MUX's and FFs) in the datapath.

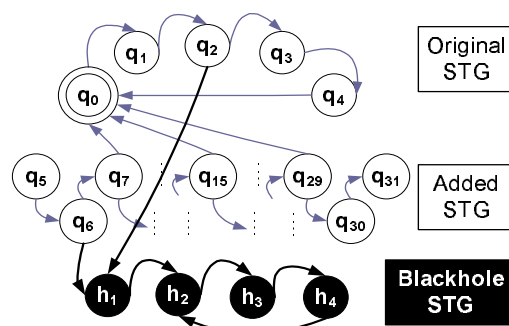


Figure 6: Example of a black hole FSM.

**(viii) Creation of identical ICs using selective IC release.** Bob only releases the ICs with similar characteristics to Alice in the hope of finding the keys by correlations. This attack is probably the most expensive because it involves only a small percentage of manufactured ICs by the untrusted foundry. Only the ICs that have similar RUBs are reported. Hence, if the attack is successful, the design house supplies many keys for ICs with similar RUBs; the birthday paradox shows that one of the keys with relatively high likelihood can be used on the unreported ICs. Note that, the way for Bob to determine closeness of characteristics is by looking at the distances of the initial power-up states.

**(ix) Differential FF activity measurement.** Bob may start to investigate the differential activities of the FFs of the unlocked designs for the same input, and then try to eliminate the FFs that have different values.

## 6.2 Countermeasures

We propose a number of mechanisms to augment the basic active metering scheme and preserve its security against the above attacks. Two important observations are that FSMs in modern industrial design are always a very small part the overall design, well below 1%, and that STG recovery is a computationally intractable problem [7, 10, 22]:

- **Creating black holes FSMs.** Alice may create a black holes FSM inside the BFSM that makes the exit impossible. Black holes are the states that cannot be exited regardless of the used input sequence. Their design is very simple as shown in Figure 6, where the black states do not have a route back to the other states. Furthermore, a designer can plan the black hole states to be permanent if it is desirable: a small part may be added, so that restarting the IC would not take it out of the black hole states. This measure essentially eliminates the effectiveness of the first two attacks, because no random input sequence leads to the initial state of the functional FSM:

once the black hole sub-FSM is entered, there is no way out. A special case is creation of trapdoor black (gray) holes FSMs that are designed in such a way that only long specific sequence of input signals known just to the designer can bring control out of this FSM and into the initial functional state of the overall FSM. An issue that needs to be carefully addressed here is preventing the IC from powering-up in one of the black-hole states. This can be easily ensured by adding extra logic to the black hole parts that would disconnect the black hole states from the power-up states.

- **Merging the functional BFSM with the test and other FSMs**, (e.g. ones that can be used for debugging and authentication). In a typical design, the functional control circuits are not the only FSMs around. Alice, with the the objective to make identification of her functional FSM more difficult, can further intricate the BFSM by co-synthesizing them with others. This augmentation makes the first two and the three CAR attacks less effective. In particular, this merger would distract the ability to simultaneously monitor the multitude of signals/values on the IC using laser reading.

- **Similar FF activity for the unlocked ICs**. The designs would be made such that once an IC exits the locked states and is in its functional states, all its FFs have a deterministic behavior that is the same for all ICs. Thus, the differential FF activity screening would not yield any useful information.

- **Creation of specialized functional FSMs (SFFSMs)**. Alice can make the security much tighter by integrating the RUBs not just to assign the initial power-up state, but to alter the structure of the BFSM and make it a SFFSM. Using this method, the reset state for FSM of each IC is a function of its RUB. Each SFFSM operates correctly only if it received a specific stream of signals from the RUBs. Since there are exponentially many states with respect to the number of FFs in FSM, we map a set of blocks that share an identical subset of RUB outputs into a single SFFSM. This countermeasure makes the first two attacks (i.e., brute-force attack and FSM reverse engineering) much more difficult and the first two CAR attacks (i.e., initial control signal CAR and initial reset state CAR) almost impossible.

A simple example of this method is presented in Figure 7. On this figure, the added STG is shown by the cloud on left, and the original STG is plotted in the right cloud. The original STG has only 3 states: a reset state two other states. Here, the original STG is replicated twice: One replication is denoted by SFFSM' and the other one is denoted by SFFSM''. The scheme adds logic to the added STG, so based on the bits in the RUB, it will be categorized into three classes. Each of the classes will

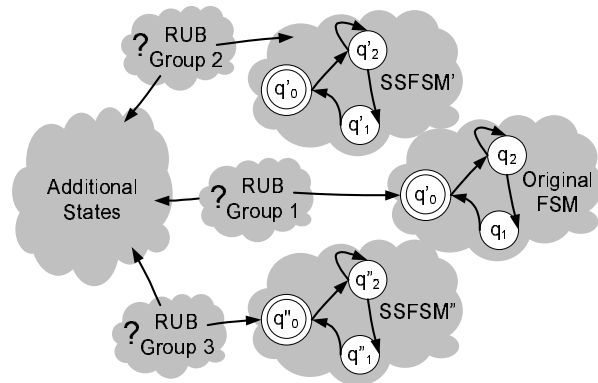


Figure 7: A simplified SFFSM.

transition to one of the reset states in one of the clouds: original FSM, SFFSM' or SFFSM''. This scheme will cause confusion in FFs scanning methods that aim at loading the reset state of an unlocked IC in the FFs of a locked chip. Note that, the replicated states need not all be unique, and maybe shared among the replicas to reduce the overhead.

The example is very small, but one can add the RUB-dependent states at various stages to ensure that the attacker is not able to break the system. A combination of the SFFSM method and the state obfuscation and encoding would ensure a full security of the design against the CAR attacks. Furthermore, using similar methods, the RUBs can be also added to the obfuscation scheme based on the dummy variables like the example presented in Figure 5, so that the same inputs would have different random obfuscation patterns.

Another use of SFFSM is for addressing the effect of temporal changes in RUB. Recall that the actual application of the new hardware metering scheme to industrial designs requires mechanisms that ensure resiliency against time-dependent permanent changes of transistors as well as gate-level and transient changes due to the environmental conditions such as temperature and supply voltage fluctuations [4].

The exact reconstruction of the first power-up state of IC (the particular one for which the designer released the key) for the purpose of defeating the variabilities is trivial: Bob can just load the captured and saved outputs of the first power-up RUB for which he has obtained the key. This mechanism makes the design susceptible to reuse attacks, where Bob can reuse the key and the initial RUB for an unlocked IC to decipher another locked IC. However, if Alice included SFFSM in her design, she would be resilient against this attack. The only technical issue that remains to be addressed is to ensure that the SFFSM receives the correct data from the physical RUB, exactly the same as the one that was first received and for

which the key is available. Otherwise, the stored key will fail.

In presence of temporal variations, ensuring that each SFFSM receives the correct data from RUB requires error-correction mechanisms. One solution is to employ standard error-correction codes (ECCs). An alternative hardware solution that encounters a lower overhead compared to ECC is to create the specifications of each SFFSM in such a way that it transitions into the correct next states, even when one or up to a specified number of the inputs from the RUB are altered by the environmental conditions. Using the hamming distances of the RUBs, we can group them into similar SFFSMs and synthesize the results such that the error correction mechanisms are inherently present. This mechanism is particularly effective for longer RUBs that are required for present industrial designs. Note that, because the minterms for the combinational logic that implements transitions are now not smaller than for non-resilient versions of the SFFSM, the hardware overhead is often zero or negative at the expense of the lower resiliency against brute force attacks [5]. However, since the probability of brute force attack can be made arbitrarily small with very low overhead (i.e., by using the black holes), this is a favorable trade-off.

- **Resiliency against combinational redundancy removal.** To overcome this attack, Alice must ensure the inapplicability of the attack to typical large circuits and the capability of this method to remove the added states. In general, computing a set of reachable states, can only be done for relatively small circuits, even when the implicit enumeration techniques are used. Thus, the method is only applicable to small circuits of small sizes.

- **Statistical characterization of gates.** Alice can go one step further and attempt to derive the gate-level characteristics of the manufactured ICs by measuring the input/output signals and exploiting the controllability and observability into the design. Essentially, knowing the circuit diagram, she would be able to write a linear systems of equations that can be solved for obtaining the approximate gate-level delay and power characteristics of the gates. She may even go further to use the extracted data to find the distribution of variations across the different chips (e.g., by using methods such as expectation maximization(EM)). Now, if the variations do not have enough fluctuations, then she will get suspicious and can halt the unlocking. This computation would ensure that the selective IC release would not be successful.

- **Obfuscation of state activities and encodings.** The implementation of the BFSM presented in the previous section renders it impossible to tell the difference between the original FSM FFs and the added states FFs.

This is because all of the FFs are changing all the time. Therefore, even though two states of BFSM in two ICs might be identical, the attacks based on scanning the FFs would not notice that, since a subset of the bits will be different. In other word, the FFs not used in the added FSM are randomly changing. Another obfuscation method that has already been implemented is that the states in the added STG are not in order and are coded out of sequence by the synthesis tool. Thus, even though there might be a direct transition (edge) between two states, the methods based on FFs readings would not notice the proximity of the two states, since there code words are distant from each other.

Note that, the attacks that were described earlier, even the ones that are computationally very expensive, will not be able to unlock the ICs, if the countermeasures described above are in place.

## 7 Experimental evaluations

To test the applicability of the method described earlier, we implemented the active hardware metering on standard benchmark designs. In this section, we present the experimental setup, followed by the overhead of implementing BFSM on the considered benchmarks. After that, we show quantitative analysis of the effectiveness of the brute force attacks. We further show how the addition of black holes can make the scheme resilient against this attack with a minimal overhead. Note that, many of the attacks described earlier are assuming structural countermeasures that are hard to quantify and evaluate.

### 7.1 Experiment setup

We used extended set of sequential benchmarks from the ISCAS'89 to evaluate the impact of the active hardware metering method [6]. Even though the ISCAS'89 benchmarks are the latest comprehensive set of the gate-level designs, they are dated compared to the complex circuits in design, production and use today. Recall that following the Moore's law, the size and complexity of the circuits doubles approximately every 18 months. We use the larger benchmarks from the set, and we project the results to more complex circuits. Our projections show that the power, area, and delay overheads diminish as we increase the size and complexity. Simultaneously, the locking complexity and resiliency against the attacks exponentially improves, due the multiplicity of states. We synthesize the benchmarks using the Berkeley SIS tool [26], that given a STG or a logic-level description of a sequential circuit produces an optimized netlist in the target technology (cell library) while preserving the sequential input-output behavior. We have written a C program that modifies the benchmarks by adding the extra states. The

program calls SIS to obtain the specifications of the synthesized and mapped original and modified STGs. When evaluating the overhead results, the important observation is that FSMs (i.e., the control circuitry) in modern industrial design are always a very small part of the overall design, well below 1% [7, 10]. Thus, even doubling the overhead, will have a minimal impact on the overall circuit that is mostly occupied by memory, testing pins, and data path circuitry.

## 7.2 Overhead of active hardware metering

Our first set of experiments study the overhead of the introduced scheme in terms of area, power, and delay. It is worth noting here that our ultimate goal is to integrate the active hardware metering method in the design flow. Thus we have considered testing the approach on manufactured ICs. However, the prohibitive cost of manufacturing a circuit in aggressive technologies (the quote we got for fabricating a circuit in 65nm was \$500K) limits our experiments to synthesizing the benchmarks. Table 1 presents the results for the area overhead. Because of the relatively small size of the circuits, we added STGs with 12 FFs and 15 FFs overhead to the original STGs. The first column shows the name of the circuit from the ISCAS'89 benchmark. The second column shows the number of inputs to the circuit. The third column shows the number of outputs to the circuits. Both the number of inputs and the number of outputs do not change after adding the extra states. The fourth column shows the number of FFs in the original circuit. The fifth column shows the area of the original circuit. Then we show both the new area and the percentage overhead after adding 12 FFs and 15 FFs for the extra states. It can be seen that the percentage area overhead is decreasing as the circuit size increases. Thus, for larger circuit sizes, the area overhead will be even less insignificant.

Table 2 shows the delay and power overheads. The first column contains the benchmark names. The second and third columns show the delay and power estimates of the original circuits. These are followed by both the delay and the percentage delay overhead, and the power and the percentage power overhead for adding both 12 FFs and 15 FFs STGs respectively. The delay overheads are universally small. With the exception of s27 that is too small to be considered practical, it is interesting to see that even other small benchmarks encountered no delay overhead after the addition of the new STG. For the small benchmarks that are not realistic compared to the current complex designs, the power increases significantly. As the circuit size increases, the percentage power overhead decreases.

Next, we make a small model of the percentage of area and power overhead versus size of the circuit to extrapolate

to more complex designs. The size of the added STG is fixed to 15 FFs. Figures 8(a) and 8(b) show the overhead data vs. size along with the fitted polynomial models, for power and area respectively. The plots suggest that as the circuit size increases, the percentage of power and area overheads both decrease. Note that, for more complex designs, it is required to add significantly more than 15 FFs. Even if adding a STG with 100 FFs would add six times the overhead of the 15 FFs case in absolute terms, the overhead would be negligible, while there will be  $2^{85}$  extra states added to the design. Thus, for current and future circuit technologies, the BFSM would have a minimal impact on the performance in terms of power, area, and delay (i.e., it will most likely stay less than 1% of the overall design).

## 7.3 Resiliency against the brute force attack

Most of the attacks described in Section 6 can be encountered by devising intelligent design strategies, as described in Subsection 6.2. The only attack that we quantitatively study here is the brute force attack. We model this attack by randomly guessing the values on the graph until arriving at the functional reset state of the original FSM.

We simulated the brute force attack on BFSMs with 12, 15, and 18 FFs, varying the inputs from 3 to 8. In this experiment, we set an upper bound of 1,000,000 guesses; if the reset state is not reached after this many trials, the original STG is considered unreachable (denoted by N/R) and the brute force attack is reported unsuccessful.

Table 3 shows the average number of guesses needed to unlock the BFSM over a 10,000 simulation runs. The first three rows show added STGs with 12, 15, and 18 FFs respectively. The next two rows show the results for STGs with 12 and 15 FFs, after adding 1 and 2 black holes respectively. Although the number of inputs does not affect the overhead, it impacts the resiliency against the brute force attack: the table illustrates that the brute force attacks are less successful if we use more than 3 different inputs. Also, as the size of the added STG increases, more guesses are necessary to unlock the circuit. By adding one black hole to the smaller FSMs, they perform better than the larger FSMs. Adding one or two black holes makes the original STG unreachable for the brute force attack. It is worth noting here that STGs with 12 and 15 FFs are really small, as they have a total of 4,096 and 32,768 states respectively. If the active metering scheme was to be implemented on current industrial strength designs, the added circuit would have at least a 100 FFs that would create  $2^{100} \sim 10^{30}$  states. It would be impossible for a brute force attack to find a key. Furthermore, addition of a few black holes will further make

Circuit	Original Details				12 FFs		15 FFs	
	In	Out	FFs	Area	Area	%	Area	%
s27	4	1	3	18	224	11.44	278	14.44
s298	3	6	14	244	454	0.86	508	1.08
s344	9	11	15	269	480	0.78	534	0.99
s444	3	6	21	352	554	0.57	609	0.73
s526	3	6	21	445	648	0.46	702	0.58
s641	35	23	17	539	743	0.38	797	0.48
s713	35	23	17	591	793	0.34	847	0.43
s953	16	23	29	743	947	0.27	1001	0.35
s832	18	19	5	769	971	0.26	1025	0.33
s1238	14	14	18	1041	1264	0.21	1318	0.27
s1423	17	5	74	1164	1382	0.19	1436	0.23
s9234	36	39	135	7971	8174	0.03	8228	0.03
s13207	31	121	453	11248	11450	0.02	11504	0.02
s38417	28	106	1463	32246	32448	0.01	32502	0.01

Table 1: Area overhead of active metering for various benchmarks.

Circuit	Original Details		12 FFs				15 FFs			
	Delay	Power	Delay	%	Power	%	Delay	%	Power	%
s27	6.60	134.00	14.40	1.18	1418.70	9.59	14.40	1.18	1696.70	11.66
s298	15.00	1167.20	15.00	0.00	2468.60	1.11	15.00	0.00	2746.60	1.35
s344	27.00	1030.00	27.00	0.00	2325.90	1.26	27.00	0.00	2603.90	1.53
s444	17.60	1550.80	17.60	0.00	2815.20	0.82	17.60	0.00	3152.30	1.03
s526	15.20	2065.70	15.20	0.00	3334.30	0.61	15.20	0.00	3664.70	0.77
s641	97.60	1560.60	97.60	0.00	2832.10	0.81	97.60	0.00	3162.40	1.03
s713	100.00	1670.70	100.00	0.00	2935.00	0.76	100.00	0.00	3265.40	0.95
s953	23.60	1816.50	23.60	0.00	3084.20	0.70	23.60	0.00	3414.60	0.88
s832	28.80	2849.60	28.80	0.00	4114.00	0.44	28.80	0.00	4444.40	0.56
s1238	34.40	2709.40	34.40	0.00	4034.00	0.49	34.40	0.00	4312.00	0.59
s1423	92.40	4882.70	92.40	0.00	6226.30	0.28	92.40	0.00	6504.30	0.33
s5378	32.20	12459.40	32.20	0.00	13515.00	0.08	32.20	0.00	14057.50	0.13
s9234	75.80	19385.50	75.80	0.00	20653.30	0.07	75.80	0.00	20983.70	0.08
s13207	85.60	37874.00	85.60	0.00	39138.40	0.03	85.60	0.00	39402.00	0.04
s38417	69.40	112706.80	69.40	0.00	113869.00	0.01	69.40	0.00	114147.00	0.01

Table 2: Delay and power overhead of active metering for various benchmarks.

the system resilient against the brute force attack.

Table 4 shows area and power overheads for adding a black hole with 2 states to added STGs with 12 and 15 FFs respectively. The overhead of adding a black hole does not exceed 5% even for very small benchmarks. For larger circuits it is unnoticeable. Note that, we often add more than one black hole to the design, to warrant the impossibility of the brute force attacks.

To evaluate the diversity of keys, we studied the number of cycles in the added STGs. For this STG, we form a new graph STG\*, that has the same nodes as STG, but reverses the edges. Note that, simultaneously reversing all the edges will not affect the number of cycles in the graph. Since each state on STG has a path to the reset state, the directed acyclic graph (DAG) rooted at the original reset state in STG\* will have a path to all states. We find a DAG of STG\* by using the Dijkstra’s shortest path

algorithm. Next, we add the STG\* edges to the DAG and see if they form a cycle and combine the cycles into one node; we iteratively continue until the cycles are gone. This approximate method is used to count the number of cycles. Using the method, we roughly guess that the STG with 12 FFs had more than 40 cycles that enables the use to build exponentially many keys for traversal from a certain state. The large number of keys can be easily generated by a combination of cycling and switching between the cycles of the STG.

## 8 Potential applications

Active hardware metering provides strong anti-piracy mechanisms for hardware IP cores as well as remote-disabling mechanisms for the manufactured parts. Remote disabling can be accomplished if a malicious activ-

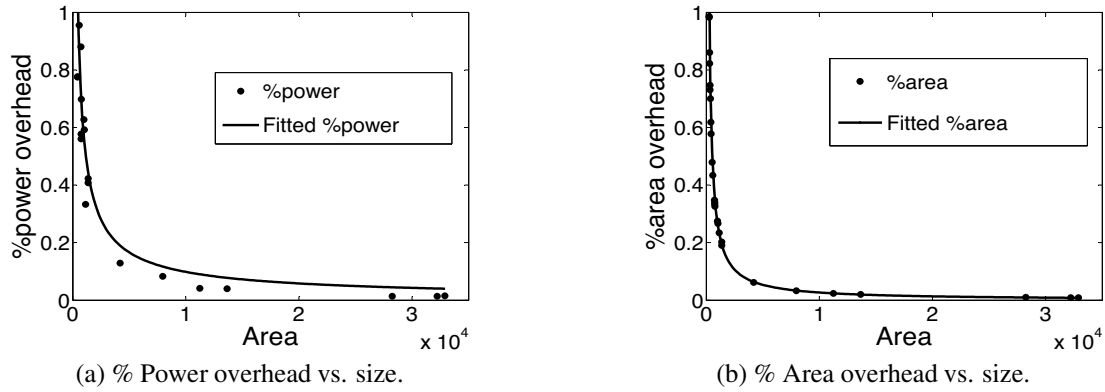


Figure 8: Percentage of (a) power; and (b) area; overheads vs. size after adding a 5 FFs STG.

bits	Number of inputs					
	3	4	5	6	7	8
<b>12</b>	74385	82708	78939	83156	77028	82490
<b>15</b>	560976	610373	602157	557776	592681	596260
<b>18</b>	933680	932501	938583	918312	N/R	N/R
<b>12 + bh</b>	998000	999000	N/R	N/R	N/R	N/R
<b>15 + bh</b>	N/R	N/R	N/R	N/R	N/R	N/R
<b>12 + 2 bh</b>	N/R	N/R	N/R	N/R	N/R	N/R
<b>12 + 2 bh</b>	N/R	N/R	N/R	N/R	N/R	N/R

Table 3: Average number of attempts needed for the brute force attack to unlock the added STG.

ity is detected. For example, a designer can add an extra part to the circuit that detects say, the brute force attack where too many invalid inputs are being entered. As another example, the strange activity patterns of the chip may be detected from a network. Upon detecting such a situation, a built-in disabling function would be invoked that transitions the IC into a non-functional state. If this state is a black hole, the IC cannot be used.

Generally speaking, combinations of the two employed security mechanisms, variability-based uniqueness of each IC, and structural manipulation of FSM while preserving the original behavioral specification, provide powerful basis for creating many security and DRM protocols. A few of the many possibilities are: (i) use of a combination of unique functionality and RUB for remote authentication and disablement of smart cards; (ii) certification that a computation was executed on a specified IC in a distributed environment; and (iii) creation of techniques to produce software than can only run on a specific IC, thereby preventing software piracy.

Furthermore, the introduced method has the potential for a broad impact on the IC industry and military use of hardware. As an example, new royalty enforcement systems can be enabled: design reuse has emerged as a dominant strategy, where different IP cores are often supplied by different vendors. The final integrator pays each

IP supplier royalties that are proportional to the number of manufactured ICs. All that is needed for royalty enforcement is that each supplier uses its own active metering scheme inside its IP.

## 9 Conclusion

We propose the first active hardware metering scheme that symmetrically protects the IP designer and the foundry by providing a key-exchange mechanism. The active metering method utilizes the unclonable variability-based ID of each silicon circuit (RUB) to uniquely lock the IC at the fabrication house. The FSM of the design is enhanced to include many added states, designed such that the RUB-based state is one of the random states with a very high probability. The state addition was done in such a way that it would not affect the functionality of the original design. The key to the locked IC can only be provided by the designer who knows the state transition graph of the design. We have illustrated the addition of black hole states to the BFSM which can be utilized for remote control and disabling of the ICs. Black hole states are also useful in making the protection scheme highly resilient against the brute force attacks. We presented a low overhead implementation for the hardware metering scheme, identified a comprehen-

Circuit	12 FFs		15 FFs	
	% Area	% Power	% Area	% Power
s27	0.05	0.04	0.04	0.03
s298	0.02	0.02	0.02	0.02
s344	0.04	0.02	0.03	0.02
s444	0.03	0.02	0.02	0.02
s526	0.01	0.02	0.01	0.02
s641	0.02	0.02	0.02	0.02
s713	0.01	0.02	0.01	0.02
s953	0.02	0.02	0.02	0.02
s832	0.02	0.01	0.02	0.01
s1238	0.01	0.01	0.01	0.01
s1423	0.01	0.01	0.01	0.01
s5378	0.00	0.00	0.00	0.00
s9234	0.00	0.00	0.00	0.00
s13207	0.00	0.00	0.00	0.00
s38417	0.00	0.00	0.00	0.00

Table 4: Percentage of area and power overheads after adding one blackhole.

sive set of possible attacks, and provided mechanisms that make the scheme much more resilient against the attacks. Experimental evaluations of the proposed metering method on standard benchmark circuits illustrate the low overhead and the applicability of the approach on industrial-size designs and its resiliency against different attacks.

## References

- [1] D.G. Abraham, G.M. Dolan, G.P. Double, and J.V. Stevens. Transaction security system. *IBM Systems Journal*, 30(2):206–229, 1991.
- [2] R. Anderson and M. Kuhn. Tamper resistance - a cautionary note. In *USENIX Workshop on Electronic Commerce*, pages 1–11, 1996.
- [3] R.J. Anderson. *Security Engineering: A guide to building dependable distributed systems*. John Wiley and Sons, 2001.
- [4] K. Bernstein, D.J. Frank, A.E. Gattiker, W. Haensch, B.L. Ji, S.R. Nassif, E.J. Nowak, D.J. Pearson, and N.J. Rohrer. High-performance CMOS variability in the 65-nm regime and beyond. *IBM Journal of Research and Development*, 50(4/5):433–450, 2006.
- [5] R.K. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.
- [6] F. Brgles, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *International Symposium of Circuits and Systems*, pages 1929–1934, 1989.
- [7] A.P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R.W. Brodersen. Optimizing power using transformations. *IEEE Trans. CAD of Integrated Circuits and Systems*, 14(1):12–31, 1995.
- [8] Defense Science Board (DSB) study on High Performance Microchip Supply. [http://www.acq.osd.mil/dsb/reports/2005-02-HPMS\\_Report\\_Final.pdf](http://www.acq.osd.mil/dsb/reports/2005-02-HPMS_Report_Final.pdf), 2005.
- [9] B. Gassend, D. Lim, D. Clarke, M. van Dijk, and S. Devadas. *Concurrency and Computation: Practice and Experience*, volume 16, chapter Identification and authentication of integrated circuits, pages 1077–1098. John Wiley & Sons, 2004.
- [10] J.L. Hennessy and D.A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers, 1996.
- [11] A. Kahng, J. Lach, W. Mangione-Smith, S. Mantik, I. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe. Watermarking techniques for intellectual property protection. In *Design Automation Conference (DAC)*, pages 776–781, 1998.
- [12] D. Kirovski, Y.-Y. Hwang, M. Potkonjak, and J. Cong. Intellectual property protection by watermarking combinational logic synthesis solutions. In *International Conference on Computer Aided Design (ICCAD)*, pages 194–198, 1998.
- [13] D. Kirovski and M. Potkonjak. Local watermarks: methodology and application to behavioral synthesis. *IEEE Trans. CAD*, 22(9):1277–1283, 2003.
- [14] F. Koeune and F. Standaert. A tutorial on physical security and side-channel attacks. In *Foundations of Security Analysis and Design (FOSAD)*, pages 78–108, 2004.
- [15] F. Koushanfar, I. Hong, and M. Potkonjak. Behavioral synthesis techniques for intellectual property protection. *ACM Trans. Design Automation of Electronic Systems*, 10(3):523–545, 2005.
- [16] F. Koushanfar and G. Qu. Hardware metering. In *Design Automation Conference (DAC)*, pages 490–493, 2001.
- [17] F. Koushanfar, G. Qu, and M. Potkonjak. Intellectual property metering. In *Information Hiding Workshop (IHW)*, pages 81–95, 2001.

- [18] J. Lach, W.H. Mangione-Smith, and M. Potkonjak. Fingerprinting digital circuits on programmable hardware. In *Information Hiding Workshop (IHW)*, pages 16–32, 1998.
- [19] J.W. Lee, L. Daihyun, B. Gassend, G.E. Suh, M. van Dijk, and S. Devadas. A technique to build a secret key in integrated circuits for identification and authentication applications. In *Symposium of VLSI Circuits*, pages 176–179, 2004.
- [20] K. Lofstrom, W.R. Daasch, and D. Taylor. IC identification circuits using device mismatch. In *International Solid State Circuits Conference (ISSCC)*, pages 372–373, 2000.
- [21] S. Maeda, H. Kuriyama, T. Ipposhi, S. Maegawa, Y. Inoue, M. Inuishi, N. Kotani, and T. Nishimura. An artificial fingerprint device (AFD): a study of identification number applications utilizing characteristics variation of polycrystalline silicon TFTs. *IEEE Trans. Electron Devices*, 50(6):1451–1458, 2003.
- [22] A. Oliveira. Techniques for the creation of digital watermarks in sequential circuit designs. *IEEE Trans. CAD of Integrated Circuits and Systems*, 20(9):1101–1117, 2001.
- [23] G. Qu and M. Potkonjak. *Intellectual Property Protection in VLSI Design*. Kluwer Academic Publisher, 2003.
- [24] S. Roy and A. Asenov. Where do the dopants go? *Science*, 309(5733):388–390, 2005.
- [25] H. Savoj and R.K. Brayton. On the optimization power of retiming and resynthesis transformations. In *Design Automation Conference (DAC)*, pages 297–301, 1990.
- [26] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.
- [27] A. Srivastava, D. Sylvester, and D. Blaauw. *Statistical Analysis and Optimization for VLSI: Timing and Power*. Series on Integrated Circuits and Systems. Springer, 2005.
- [28] Y. Su, J. Holleman, and B. Otis. A 1.6J/bit stable chip ID generating circuit using process variations. In *International Solid State Circuits Conference (ISSCC)*, page to appear, 2007.
- [29] G.E. Suh, C.W. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *International Symposium on Computer Architecture (ISCA)*, pages 25–36, 2005.
- [30] I. Torunoglu and E. Charbon. Watermarking-based copyright protection of sequential functions. *IEEE Journal of Solid-State Circuits (JSSC)*, 35(3):434–440, 2000.
- [31] VSI Alliance - Intellectual Property Protection Development Working Group, “White Paper: The Value and Management of Intellectual Assets”. <http://vsi.org/documents/datasheets/TOC.IPPWP210.pdf>, 2002.
- [32] J.L. Wong, R. Majumdar, and M. Potkonjak. Fair watermarking using combinatorial isolation lemmas. *IEEE Trans. CAD*, 23(11):1566–1574, 2004.

## Notes

<sup>1</sup>In this paper, the term IP is used to refer to the integrated circuits design specifications that is available to the fabrication house.

# On Attack Causality in Internet-Connected Cellular Networks

Patrick Traynor, Patrick McDaniel and Thomas La Porta  
Systems and Internet Infrastructure Security Laboratory  
Networking and Security Research Center  
The Pennsylvania State University  
University Park, PA 16802  
{traynor, mcdaniel, tlp}@cse.psu.edu

## Abstract

*The emergence of connections between telecommunications networks and the Internet creates significant avenues for exploitation. For example, through the use of small volumes of targeted traffic, researchers have demonstrated a number of attacks capable of denying service to users in major metropolitan areas. While such investigations have explored the impact of specific vulnerabilities, they neglect to address a larger issue - how the architecture of cellular networks makes these systems susceptible to denial of service attacks. As we show in this paper, these problems have little to do with a mismatch of available bandwidth. Instead, they are the result of the pairing of two networks built on fundamentally opposing design philosophies. We support this claim by presenting two new attacks on cellular data services. These attacks are capable of preventing the use of high-bandwidth cellular data services throughout an area the size of Manhattan with less than 200Kbps of malicious traffic. We then examine the characteristics common to these and previous attacks as a means of explaining why such vulnerabilities are artifacts of design rigidity. Specifically, we show that the shoehorning of data communications protocols onto a network rigorously optimized for the delivery of voice causes that network to fail under modest loads.*

## 1 Introduction

The interconnection of cellular networks and the Internet significantly expands the services available to telecommunications subscribers. Once limited to basic voice services, these systems now offer data connections at the lower end of broadband speeds. Accordingly, devices attached to such networks are capable of engaging in applications ranging from traditional voice communications to streaming video. While initial uptake of these services has been slow [1, 18], notable advances in connection speed and an expanded set of supported devices

(e.g., laptops) are beginning to spur substantial acceptance and usage.

The transformation of these systems from isolated providers of telephony to Internet-attached general purpose communication networks has already been marred by concerns of inadequate security. As connections between such systems and external data networks have developed, a number of researchers have noted weaknesses in the telecommunications infrastructure. For example, our previous work on targeted text messaging attacks demonstrated the ability to deny service to large metropolitan areas with the bandwidth available to a single cable modem [16, 47]. While these and a host of other exploits [39, 44] have explored the impact of specific attacks against cellular networks, they have all failed to answer a larger question: “How does the architecture of cellular data networks inherently make them susceptible to denial of service attacks?” Unexpectedly, the answer to this question has little to do with bandwidth constraints. Instead, these vulnerabilities are the result of the conflict caused by connecting two networks built on fundamentally opposing design philosophies.

In this paper, we argue that low-bandwidth denial of service attacks in telecommunications networks are artifacts of incompatibility caused by interconnecting systems built with two differing sets of design requirements. While the merits of independent “smart” and “dumb” architectures have been widely debated, none have examined the inherent security issues caused by the connection of two mature systems built on these opposing design tenets. To support our assertion, we present two new vulnerabilities in cellular data services. These attacks specifically exploit connection setup and teardown procedures in networks implementing the General Packet Radio Service (GPRS). Through a combination of analysis and simulation, we characterize the impact of such attacks on legitimate voice and data services in the network. We then use these new attacks, in combination with previously discussed vulnerabilities, as demonstra-

ble evidence that the translation of traffic between these two network architectures is the root of such problems. Through this, we seek to develop a larger sense for *why* such attacks are possible, even in the presence of a cellular network with hypothetically infinite bandwidth. Ultimately, by understanding causality, the discovery of future vulnerabilities is vastly simplified.

In so doing, we make the following contributions in this work:

- **New Vulnerability Analysis:** We identify and develop a realistic characterization of two new vulnerabilities in cellular data networks. These exploits target specific components of the expensive connection setup and teardown procedures and can prevent legitimate use of data services. While the partitioning of voice and data flows in such networks is designed to protect each traffic type from the other, our attack on setup mechanisms demonstrates that optimizations made for efficiency can result in the disruption of voice services.
- **Implications of Combined Design Philosophies on Security:** We use the body of available vulnerabilities as the basis for an analysis to determine the underlying cause of such denial of service attacks. Consequently, we show that these problems are not necessarily the result of poor protocol design but are instead deeply rooted in opposing architectural assumptions.

The remainder of this paper is organized as follows: Section 2 offers a brief overview of our previous work on targeted SMS attacks to prime the reader with additional data points; Section 3 presents and offers an initial analysis for our newly discovered vulnerabilities; Section 4 uses monitoring of deployed cellular networks and simulation to support the conclusions made in the previous section; Section 5 coalesces the previous attacks on cellular networks as data points in our larger argument; Section 6 offers a discussion of techniques to address such problems; Section 7 provides related work; Section 8 offers concluding thoughts.

## 2 Prior Work - Text Messaging Attacks

We present a high-level overview of our previous attacks on text messaging [16, 47]. With some five billion messages sent each month in the United States alone [28], this service has become one of the premier streams of revenue for cellular network operators. To encourage widespread use, providers have opened a significant number of gateways between the Internet and their networks. Whether through email, instant messaging applications or even a provider's website, it is possible to ex-

change asynchronous communications with cellular subscribers. The ability to communicate across such networks, however, is not without potential consequences.

A cellular network<sup>1</sup> must perform multiple tasks before delivering a text message. The network first conducts a series of lookups to determine the location of the destination device. The device must then be awoken from an energy-saving sleep state and authenticated. A connection can then be established and the incoming text message delivered. Critical to this process is the *Standalone Dedicated Control Channel* (SDCCH), which is responsible for the authentication and content delivery phases of text messaging. With a bandwidth of 762bps [6], this constrained channel is shared by the setup phases of both text messaging and voice calls. Consequently, by keeping the SDCCH saturated with text messages, incoming legitimate voice and text messages can not be delivered by the network. Understanding this, an adversary attempting to exploit this system can use web-scraping and feedback from provider websites to create "hit-lists" of targeted devices. By sending traffic to these targeted devices at a rate of approximately 580Kbps, the adversary would be able to deny service to all of Manhattan.

Attack mitigation techniques, ranging from queue management to resource allocation strategies on the air interface, were then shown to diminish much of the impact of such attacks. While successful, these countermeasures did not consider the use of cellular data services such as GPRS to alleviate targeted text messaging attacks. Logically, delivering data traffic over separate, higher bandwidth links should provide the most complete solution to this problem. However, as we show in the next section, it is possible to disrupt cellular data services with less bandwidth than was used in the original SMS attack.

## 3 New Vulnerabilities in Cellular Data Services

We present two new denial of service (DoS) vulnerabilities in cellular data services. These attacks use a relatively small amount of traffic to exploit connection setup and teardown mechanisms. We use publicly available specifications to provide an initial characterization of these attacks and as a means of demonstrating the potential for the interruption of data services in major metropolitan areas.

### 3.1 Network Architecture

Before a GPRS/EDGE<sup>2</sup> network provides any services to a mobile device user, a series of attachment and authentication procedures must take place. On power-up,

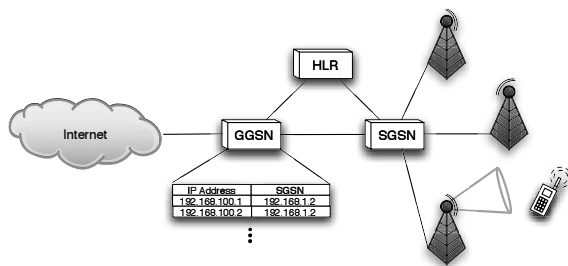


Figure 1: A high level network architecture for cellular data networks.

a device (e.g., mobile phone) transmits a *GPRS-attach* message to the network. The base station forwards this message to the attached *Serving GPRS Support Node* (SGSN), which authenticates the user's identity with the help of the *Home Location Register* (HLR). The HLR supports both voice and data operations in the network by keeping track of information including user location, availability and accessible services. When this process completes, the mobile device has a virtual connection with the network.

In order to exchange packets with external networks, the mobile device must then establish a *Packet Data Protocol* (PDP) context with the network. The PDP context is a data structure stored in the SGSN and the *Gateway GPRS Support Node* (GGSN) and is responsible for mapping billing information, quality of service requirements and an IP address to a user device. While many phones do not currently automatically establish a PDP context on power-up, the trend towards doing so (e.g., email-capable phones and GPRS-equipped laptops) is rapidly increasing. As cellular providers move into the broadband Internet market, such numbers will continue to expand rapidly.

Having been authenticated and registered, a mobile device is capable of exchanging packets with hosts internal and external to the cellular network. At some time after attachment, a packet originating from an Internet-based host and destined for a mobile device arrives at the GGSN. The GGSN compares the destination IP address to those of established PDP contexts and, upon finding the corresponding entry, forwards the packet to the corresponding SGSN. The SGSN begins the process of connection establishment and wireless delivery. Figure 1 highlights this network architecture.

The final hop of packet delivery occurs over the air interface. The details of this step, however, depend upon the current state of the device. As power has tradition-

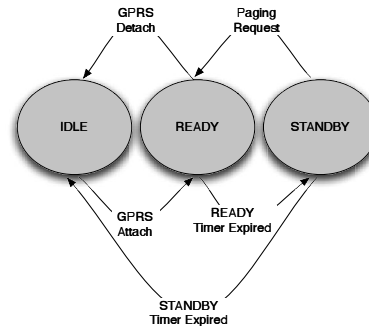


Figure 2: A state transition diagram for mobile devices, including transition functions.

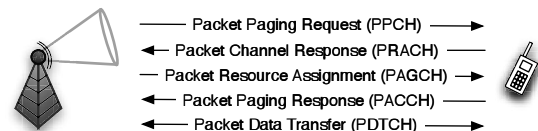


Figure 3: When the first packet of a session arrives at the base station, the host must be paged and then assigned logical resources. The messages and channels used to accomplish this are shown above.

ally been a concern in this setting, mobile devices are not constantly listening for incoming packets. To accommodate this constraint, devices operate in one of three states: IDLE, STANDBY, and READY. Devices in the IDLE state are unregistered with the network and therefore unreachable. In the power-saving STANDBY state, in which the vast majority of time is spent, devices periodically listen for network “wake up” messages known as pages. Upon receiving a page from the network, the device transitions into the READY state. In this state, a device constantly monitors the air interface for incoming packets. When packets are not received for a number of seconds, devices transition back into the STANDBY state to conserve power. These three states and the transitions between them are shown in Figure 2.

On the arrival of the first packet in a flow, the SGSN begins the process of locating the targeted device. If the destination device is not currently in the READY state, the base station nearest to the device is unknown to the network. Accordingly, the SGSN creates paging messages to be sent from a number of base stations. Upon receiving a paging request, a base station transmits a message to multiple sectors (i.e., service areas) over the *Packet Paging Channel* (PPCH). Whether due to interference or sleep cycles, the paging process typically re-

quires multiple iterations. If the targeted device is awake and hears its temporary identifier in a paging message, it attempts to alert the network of its presence by responding on the *Packet Random Access Channel* (PRACH). The base station receiving this response alerts the SGSN that the destination device has been located. The network then responds on the *Packet Access Grant Channel* (PAGCH) with a message containing a list of *Packet Data Traffic Channels* (PDTCHs) that should be monitored for incoming data. The device acknowledges receiving this message over the *Packet Associated Control Channel* (PACCH). At the end of this setup, as illustrated in Figure 3, the network can then route traffic directly to the READY state device. Note that the above channels are largely complementary to channels used for voice signaling (the naming convention, minus the “Packet” prefix, is the same). Because running two sets of control channels leads to the underuse of limited spectrum, the standards documents indicate that it is acceptable for voice and data control channels to be shared [3, 7].

### 3.2 Packet Multiplexing on the Air Interface

Data services have been available from cellular networks for a number of years. Like voice telephony, these circuit-switched services required that a single endpoint monopolize a channel for the entire duration of its connection to the network. Regardless of whether this connection was used to constantly stream content or intermittently deliver packets, the provider charged the end user for the entire duration of the connection. Accordingly, demand for such inefficient services was not great. GPRS overcomes these limitations by multiplexing multiple traffic flows over individual links. Accordingly, it is possible to serve a large number of users on a single physical channel concurrently and only charge them for the packets they exchange.

GPRS provides data service by building on the timeslot structure of GSM. Specifically, a contiguous piece of radio spectrum is subdivided into equal timeslots. When assigned a timeslot, a user exerts temporary control over a small piece of the air interface. To provide the illusion of continuous control, sets of eight timeslots are grouped into a frame so that each can be serviced once every 4.615ms. This sampling across timeslots creates physical channels, upon which voice, data and control traffic can be delivered. When used for data, these physical channels are referred to as *Packet Data Channels* (PDTCHs). Each set of 52 frames creates larger units known as multiframe. These multiframes are subdivided into 12, four-timeslot blocks, with logical channels then mapped onto each block. The remaining four timeslots in a multiframe are used for time synchronization and signal strength

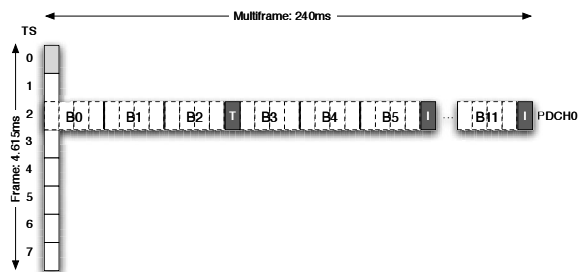


Figure 4: Each timeslot in a GPRS TDMA frame is used to create physical channels called Packet Data Channels (PDTCHs). Every 52-frame time period creates a multiframe, which is divided into twelve bursts of four. Each group, or bursts, holds a single logical channel. The specific allocation of these channels is dependent on the network. The remaining timeslots are used for time synchronization and idle measurement.

measurement periods. For example, in Figure 4, block B0 may function as a PPCH and blocks B1, B4 and B7 may be used as PDTCHs<sup>3</sup> [7].

When the first packet in a flow arrives at a base station for a user in STANDBY mode, the paging method described above occurs. As part of connection establishment, the flow receives a unique MAC layer label known as the *Temporary Flow Identifier* (TFI). Every subsequent packet belonging to the *Temporary Block Flow* (TBF) is marked with this TFI so that a targeted mobile device knows which packets to decode. When the base station has no more packets to send to the destination mobile device, the TBF and its associated TFI expire and can be reused by other flows in the immediate area. Upon TBF expiration, the mobile device returns to the STANDBY state.

### 3.3 Exploiting Teardown Mechanisms

Because the process of locating, paging and establishing a connection between the network and an end device is expensive, the immediate expiration of a TBF is impractical. For example, minor variations in packet interarrival times would force a system as described above to frequently relocate, repage and reestablish connectivity with users. Accordingly, networks implement a delayed teardown of resources. This means that devices remain in the READY state and retain their TBF for a number of seconds before the network attempts to reclaim its logical resources. When a packet is delivered to the user, the network sets a timer<sup>4</sup>, which is reset to its default value on the arrival of each additional packet. The standards recommend a timer value of approximately five seconds [2]. Given that the connection establishment process requires roughly the same amount of time, such a value is entirely reasonable.

Because TFIs are implemented as a 5-bit field, an adversary capable of sending 32 messages to each sector in a metropolitan area can exhaust logical resources and temporarily prevent users from receiving traffic. Targeted devices would not need to be infected or controlled by the adversary; rather, hit-list generation techniques similar to those discussed in our previous work [16] could be used to locate hosts able to receive traffic. If this task can be repeated before the TBF timers expire, a denial of service attack becomes sustainable. In order to more explicitly characterize the bandwidth requirements, we model such an attack on Manhattan using well known parameters [35, 48]. Given an area of 31.1 miles<sup>2</sup> and a sector coverage area of approximately 0.5 and 0.75 miles<sup>2</sup>, Manhattan contains 55 sectors. Using a READY timer of 5 seconds and 41 byte attack packets (i.e., TCP/IP headers plus one byte), the delivery of legitimate data services in Manhattan could be prevented with the attack shown below:

$$\begin{aligned} \text{Capacity} &\approx 55 \text{ sectors} \times \frac{32 \text{ msg}}{1 \text{ sector}} \times \frac{41 \text{ bytes}}{1 \text{ msg}} \times \frac{1}{5 \text{ sec}} \\ &\approx 110 \text{ Kbps} \end{aligned}$$

The exhaustion of all hypothetical TBFs may not be necessary given current usage and deployed hardware. As the current demand for voice services far outpaces cellular data usage, only a small percentage of physical channels in a sector are used as PDCHs. Because GPRS/EDGE are not extremely high bandwidth services, allowing 32 individual flows to be concurrently multiplexed across a single PDCH would be detrimental to individual throughput. Accordingly, often only a subset of the 32 TBFs (4, 8 or 16 [26, 33]) are usable. The maximum number of concurrent TBFs in a sector is therefore  $\min(d * u, 32)$ , where  $d$  is the number of downlink PDCHs and  $u$  is the maximum number of users per PDCH. While the number of PDCHs can be dynamically increased in response to rising demand for data services, networks typically hold unused channels to absorb spikes in voice calls. It is therefore unlikely that all 32 TBFs will be available at all times, if ever. A more realistic approximation of the bandwidth required to deny access to data services is given by:

$$\begin{aligned} \text{Capacity} &\approx 55 \text{ sectors} \times \frac{4 \rightarrow 16 \text{ msg}}{1 \text{ sector}} \times \frac{41 \text{ bytes}}{1 \text{ msg}} \times \frac{1}{5 \text{ sec}} \\ &\approx 14.1 \rightarrow 56.4 \text{ Kbps} \end{aligned}$$

The brute-force method of attacking a cellular data network in a metropolitan setting is simply to saturate all of the physical channels with traffic. Even at their greatest levels of provisioning, the fastest cellular data services are simply no match against traffic generated by

Internet-based adversaries [39, 45]. Such attacks, obvious by the sheer volume of traffic created, would likely be noticed and mitigated at the gateways to the network. However, with knowledge of the interaction between different network elements, it is possible for an adversary to launch a much smaller attack capable of achieving the same ends. A basic understanding of the packet delivery process provides the requisite information for realizing this attack.

Given a theoretical maximum capacity of 171.2 Kbps per frequency and as many as 8 allocated frequencies per sector, an adversary attempting the brute-force saturation of such a system would instead need to generate the volume of traffic as calculated as:

$$\begin{aligned} \text{Capacity} &\approx 55 \text{ sectors} \times \frac{171.2 \text{ Kbps}}{1 \text{ frequency}} \times \frac{8 \text{ frequencies}}{1 \text{ sector}} \\ &\approx 73.56 \text{ Mbps} \end{aligned}$$

By attacking the logical channels instead of the raw theoretical bandwidth, *an adversary can reduce the amount of traffic needed to deny service to a metropolitan area by as much as three orders of magnitude*. Note that networks implementing EDGE, which can provide three times the bandwidth of a GPRS system, would experience the same consequences given the same volume of attack traffic.

### 3.4 Exploiting Setup Procedures

If connections to an end host must repeatedly be reestablished, the interarrival time between successive packets becomes exceedingly large. Delaying resource reclamation is therefore a necessary mechanism to ensure some semblance of continuous connectivity to the network. This latency, however, is not simply the result of the time required for a user to overhear an incoming paging request. To better understand setup cost, we examine a network in which resource reclamation occurs immediately after the last packet in a flow is received.

Of particular interest to such an analysis is the performance of the common uplink channel, the PRACH. Because this channel is shared by all hosts attempting to establish connections with the network, the PRACH inherently has the potential to be a system bottleneck. To minimize contention, access to the PRACH is mediated through the slotted-ALOHA protocol. Given a channel divided into timeslots of size  $t$  and time synchronization across hosts, end devices attempting to establish connections transmit requests at the beginning of a timeslot. In so doing, the network reduces the amount of time during which collision can occur from  $2t$  in the random access case to  $t$ . While slotted-ALOHA offers a significant improvement over random access, its throughput remains

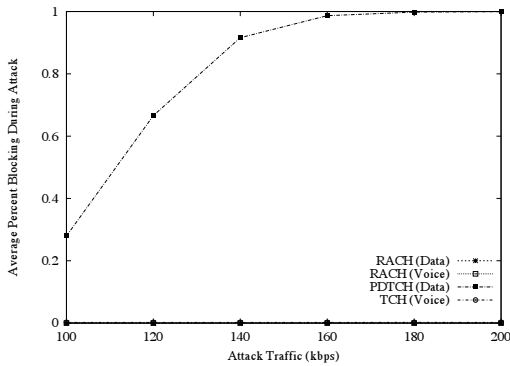


Figure 5: Blocking of legitimate traffic for varying attack traffic loads. Note that blocking only occurs on the PDTCH. These loads represent the entire attack bandwidth used across Manhattan.

low. Given a traffic intensity of  $G$  messages per unit time, the normalized throughput  $\gamma$  of slotted-ALOHA is:

$$\gamma = Ge^{-G}$$

The maximum theoretical utilization of channel implementing slotted-ALOHA is 0.368. In reality, however, this value is significantly lower. As the number of incoming connection establishment requests increases, so too does the need for retransmission due to collision. The throughput of such a system therefore typically stabilizes at a point far below this optimum value. Given a large number of paging requests, potentially caused by the immediate reclamation of resources as described above, the throughput of this already constrained channel would be severely degraded. Accordingly, the rate at which responses to connection establishment requests will pass through this channel is much lower than the available bandwidth. Because the behavior of the PRACH is highly unstable and affected by feedback (i.e., retransmissions due to collision), we leave the characterization of specific traffic volumes necessary to cause blocking to the next section.

## 4 Attack Characterization

In order to better characterize the observations made in the previous section, we extend the GSM simulator from our previous work [47] to include support for GPRS data services. The parameters of this simulator were set by information from a variety of sources. The means by which these parameters were chosen are discussed in the Appendix.

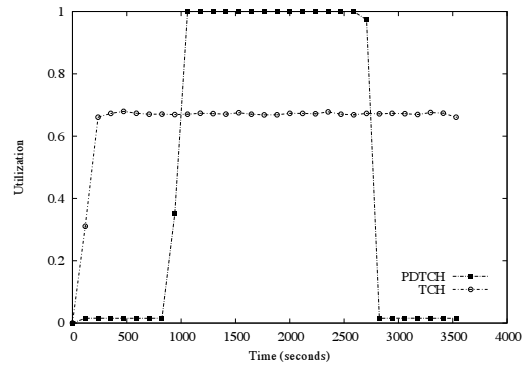


Figure 6: TFI utilization for a Manhattan-wide attack at 200Kbps. Actual PDTCH utilization (not shown) is virtually zero because of infrequent arrivals for these established flows.

### 4.1 Modeling Attacks on Teardown Mechanisms

To demonstrate the exploitation of delayed resource teardown, we simulate a GPRS network under varying traffic loads. Although the full complement of TBFs may not be available in all real deployments [26, 33], we conservatively allow for up to 32 concurrent flows. When in use, each TFI is held for exactly five seconds unless a new packet arrives. While it is possible for a single device to obtain multiple TFIs, we assume that all incoming flows for a given destination share a single TBF [4]. Because of observations made on deployed networks, both voice and data setup requests share a number of control channels. We therefore replace data control channels with their voice equivalents (i.e., RACH instead of PRACH).

Legitimate voice and data calls were modeled as Poisson random processes and generated at rates of 50,000 and 20,000 per hour, respectively, across Manhattan. The duration of these flows are also generated in a similar fashion with means of 120 and 10 seconds, respectively. These values represent standard volumes and exhibit no blocking. Attack flows, each consisting of a single packet, are also modeled by a Poisson random process with rates ranging from 100-200 Kbps. Each run, of which there were 1000 iterations for each attack load, simulated an hour of time with attacks occupying the middle 30 minutes.

Figure 5 shows the blocking rates of legitimate traffic caused by an attack on the delayed teardown mechanism. At a rate of 160 Kbps or greater, the ability to use cellular data services within Manhattan is virtually nonexistent. The amount of traffic required to execute such an attack is slightly greater than the estimation of a perfect

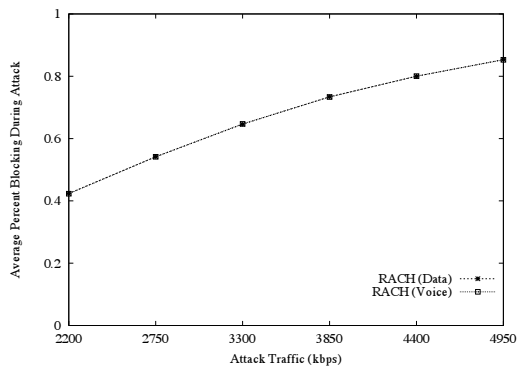


Figure 7: Blocking caused when immediate resource reclamation is enforced on data sessions. Notice that because both voice and data flows use the RACH, increased data requests cause voice blocking. No blocking was observed on other channels.

scenario in Section 3.3 due to the exponential interarrival rate used to generate packets. However, because this more realistically represents the nature of packet delivery in a network given the presence of other traffic, it offers a more accurate characterization of the attack. In spite of having the potential to deliver large volumes of traffic once flows are established, these results demonstrate that use of cellular data services can in fact be denied with less bandwidth than was used in the targeted text messaging attacks [16, 47].

Figure 6 offers additional insight into the attack by providing the utilization profile for a number of channels. Most importantly, only the PDTCHs operate at capacity during the attack. This utilization represents the state of virtual resources, not channel bandwidth. None of the channels responsible for delivering voice, most critically the *traffic channels* (TCHs), are measurably affected by the increase in data traffic. Note that this is deliberate as cellular data services such as GPRS are designed to completely separate voice and data services.

## 4.2 Modeling Attacks on Connection Setup

To characterize the impact of frequent connection reestablishment on a cellular data network, we simulate a variety of traffic levels in the presence of immediate resource recovery. Specifically, when the base station no longer has packets to send for a particular flow, the targeted device returns to the STANDBY state. Except for delayed teardown procedures, all network settings and conditions including legitimate traffic volumes and interarrival patterns, remain the same. Attacks in this scenario, each of which occurs according to a Poisson

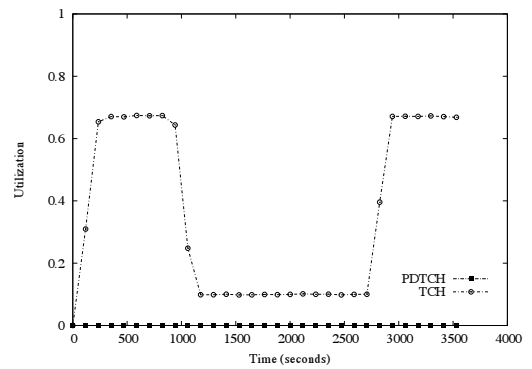


Figure 8: The impact of RACH congestion on voice calls. Notice that during the attack phase, voice call blocking on the RACH causes a significant under utilization of traffic channels.

random distribution, range from 2200-4950 Kbps spread across all of Manhattan. As in our previous experiments, each attack traffic level was run for 1000 iterations.

Figure 7 shows the blocking rates for legitimate traffic on a number of channels. Unlike the attack in the previous section, in which PDTCH blocking occurred because of TBF exhaustion, no loss of packets was observed on the PDTCHs. In spite of this, the results of these simulations confirm a more significant vulnerability - both voice and data flows experience blocking on the RACH. Although such networks strive to separate voice and data traffic, the dual use of control channels allows misbehavior in one realm to affect the other. Generating just over 3 Mbps of traffic for the entire city of Manhattan, an adversary is capable of blocking nearly 65% of all traffic - voice and data. For a network in which a blocking probability of 1% is typically viewed as unacceptable, such an attack represents a serious operational crisis.

Figure 8 provides further information about the impact of the 4950Kbps attack on voice and data services. The most notable consequence of this attack is observable in the nearly 80% decrease in TCH utilization. The near zero utilization of PDTCHs offers an explanation to the lack of blocking observed in the previous figure - the majority of legitimate traffic is being filtered out before it can ever be delivered by the PDTCHs. Accordingly, a network using the settings described above is subject to attacks capable of denying both voice and data services.

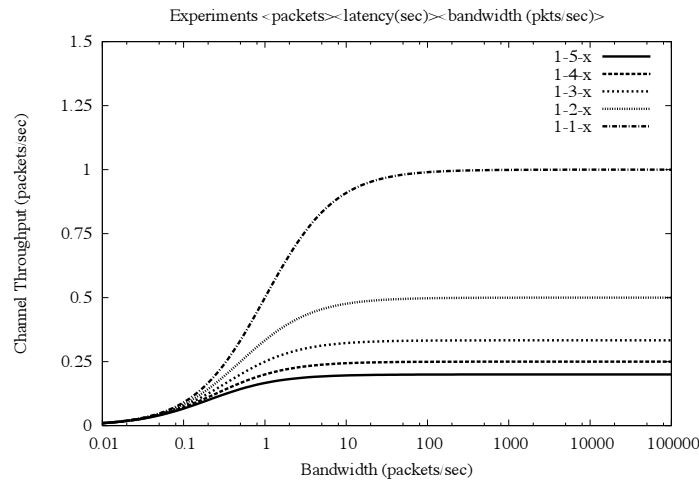


Figure 9: Given a connection establishment latency and the size of requests (in packets), we examine the impact of varying bandwidth on system throughput. When the available bandwidth allows for the virtually instantaneous delivery of requests, system throughput plateaus. This result indicates that bandwidth is ultimately not the bottleneck in this system. (log-scale)

## 5 The Meeting of Conflicting System Design Philosophies

At first glance, the differences between each of the attacks on cellular networks appear stark. Targeted text messaging attacks fill and maintain a low-bandwidth control channel at capacity. Adversaries attacking cellular data services exhaust virtual resources or take advantage of access protocol inefficiencies. In reality, all of these vulnerabilities are remnants of a conflict between the design philosophies of telecommunications and traditional data networks. Specifically, they are the result of contrasting definitions of a flow and the role of networks in establishing them. To make such a claim more concrete, we begin by demonstrating how a pair of seemingly adequate techniques for mitigating the above attacks fails to do so.

The most obvious approach to addressing the data attacks described in Section 3 is to expand the range of possible TFI values. Unfortunately, as mentioned earlier, these limitations are necessary given the bandwidth available to GRPS/EDGE networks. The use of 32 (or fewer) concurrent flows per sector is a requisite concession for providing basic levels of connectivity between the network and end devices. In order for an increased pool of identifiers to have a meaningful effect, the bandwidth available to data services would also need to be significantly increased. This combination of approaches is actually implemented in 3G cellular networks such as UMTS [8]. However, even these networks suffer from the high cost of connection establishment (i.e., delivering the first packet in a flow).

A session establishment period lasting a few seconds represents only a small fraction of the total lifetime for a connection persisting for a number of minutes. Given the limited amount of spectrum allocated to cellular providers, such infrequently used channels predictably occupy as little space as possible to avoid wasting bandwidth. Because the duration of a packet flow may not provide sufficient time over which such an expense can be amortized, the minimal allocation of bandwidth to connection establishment may in fact create a system bottleneck. To capture the impact of additional bandwidth on connection setup, we offer a simple model of request throughput for a sector as follows:

$$\text{Throughput} = \frac{\# \text{ Packets}}{\text{Setup Latency} + \frac{\# \text{ Packets}}{\text{Bandwidth}}}$$

If the expense associated with connection establishment was the result of inadequate resources, an increase in bandwidth should alleviate much of this cost. Such a scenario would be equivalent to increasing the size of the smallest link in a traditional data network to improve end-to-end throughput. However, the calculated effects of increased bandwidth on overall throughput are extremely limited in this setting. Because connection establishment exchanges contain fixed-length messages and not the variably sized packets of data delivery, the presence of additional bandwidth does little to improve performance after each channel can send paging requests instantaneously. As is shown in Figure 9, the limit of system throughput as bandwidth approaches infinity becomes:

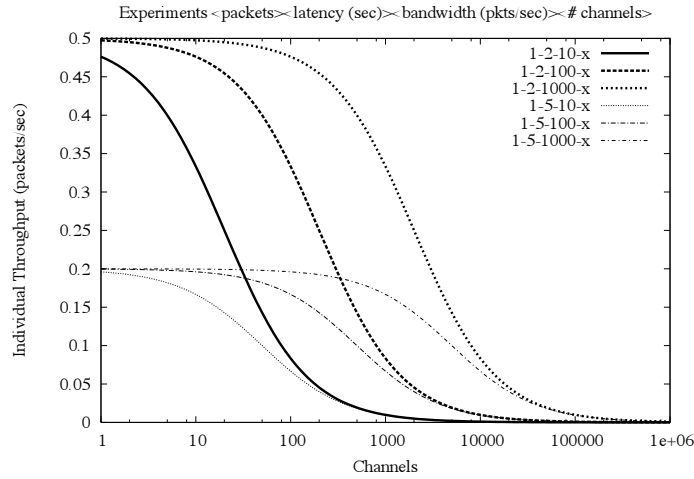


Figure 10: Increasing the number of channels can improve overall system throughput. However, individual throughputs and connection setup times react inversely. Reducing the expense of connection establishment must therefore come from a reduction in connection setup latency. (log-scale)

$$\lim_{BW \rightarrow \infty} \text{Throughput} = \frac{\# \text{ Packets}}{\text{Setup Latency}}$$

Increasing system throughput can, for this reason, be accomplished in one of two ways. In the first, the number of channels over which connections can be sent could be increased. Such a change would allow many more connection establishment requests to be sent in parallel. While increasing the throughput of the system as a whole, this approach would prove detrimental to individual users. As shown in Figure 10, subdividing a fixed bandwidth into additional channels intuitively reduces the throughput of a single user. Adding extra channels could also potentially create elevated contention for the shared uplink channel (RACH). More importantly, increasing the throughput of the system does not necessarily reduce cost with respect to delay experienced by individual users. Therefore,

*Decreasing the cost of connection establishment in a cellular data network is not a matter of increasing bandwidth but rather the reduction of connection setup latency.*

The concept of connection establishment is considerably different in cellular and traditional data networks. In the case of the former, the network must page, wake, and negotiate with a targeted device before ultimately delivering traffic. Whether due to misaligned sleep cycles,

missed paging messages or congestion, this set of operations can require more than five seconds before being able to transmit data. As discussed in Section 3, these concessions are made because the network assumes that end devices are limited both in terms of power and computational ability. True packet-switched networks provide no such services; rather, higher layers in the protocol stack implement functionality as needed. In general, each packet is treated as an individual entity and is simply forwarded to the next logical hop. Whether it is wired or wireless in nature, there is no connection to be established from the perspective of the network<sup>5</sup>. Nodes responsible for routing packets do not assume that their next hop neighbors have any specific abilities other than moving the packet closer to its intended destination. Accordingly, connection setup latency is more accurately depicted as propagation delay from the viewpoint of these networks. Given that the delay of propagation time and connection establishment differ by many orders of magnitude, the underlying cause of low-bandwidth attacks on cellular data networks becomes more clear.

The vulnerable components in both the targeted text messaging and cellular data service attacks are those mechanisms responsible for translating traffic from one network architecture to another. While a data network simply forwards individual packets as they arrive, a cellular data network interprets the first packet in a flow as an indicator of more traffic to come. Rather than simply forward that packet to its final destination, the network dedicates significant processing and bandwidth re-

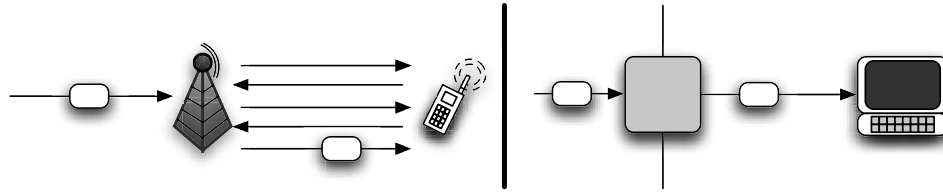


Figure 11: A comparison of the cost of delivering a single packet in cellular and traditional data networks. In the cellular data case (left), a significant amount of delay is added because of connection establishment procedures, whereas the router in the traditional setting (right) simply forwards the packet to the final hop.

sources to ensure that the end device is ready to receive data. This assumption is valid in traditional telephony because of the nature of voice communication. Except for cases of an immediate hangup, sessions are guaranteed to contain multiple “packets” of information. Data communications, however, do not necessarily share this characteristic. Any protocol or application generating packets separated by a number of seconds (e.g., instant messaging programs, session keep-alive messages, applications implementing Nagle’s algorithm [34]) violates this model. Whether it is embodied by text messages or data traffic, the amplification of a single incoming packet into a series of expensive delay inducing setup operations is the source of such attacks. Figure 11 reinforces this conclusion by comparing generalizations of the two architectures.

Connection establishment in cellular and traditional networks are so different because the philosophies upon which these systems are based are incompatible. The notion that the middle of a network provide only a limited set of simple functions is at the core of the end-to-end principle [42]. By making no assumptions about the context in which a packet’s contents will be used, the network is free to specialize in a single task - moving data. Services not used by all applications, including reliable delivery, content confidentiality and in-order arrival, become the responsibility of higher layers of the protocol stack in the end hosts. The concentration on sending packets allows networks built according to the end-to-end principle to be flexible enough to support new application types and usage models as they emerge. Telecommunications networks are built on the opposite model. Hard service requirements, especially for real-time interaction, forced the network to provide the majority of service guarantees. Because the functionality of the network was once limited to voice applications, telecommunications systems could be tightly tailored to a specific set of constraints. The inclination to build a network in such a manner was addressed by the original end-to-end argument:

*“Because the communications subsystem is frequently specified before the applications*

*that use the subsystem are known, the designer may be tempted to “help” the users by taking on more function than necessary.” [42]*

Because these specialized networks implement more functionality than is absolutely necessary, they exhibit *rigidity*, or the inability to adapt to meet changing requirements or usage [15]. Rigidity in design causes such systems to enforce assumptions appropriate for one subset of traffic on all others. The treatment of each packet as part of a larger flow is one embodiment of such inflexibility. This rigidity is also apparent when examined from the perspective of evolving end devices. For example, many laptops now contain hardware supplying access to cellular data networks [21, 37]. Regardless of their ability to implement services at higher layers of the protocol stack or their access to power, these end devices are forced to transition between STANDBY and READY states simply because such behavior is mandated by the network. Devices connecting via 802.11 could simply trade off the overhead associated with paging at the cost of additional power use. This point is made more obvious when put in the context of home or office LANs supported by a cellular backhaul connection. The network would require such systems to participate in the process of location determination and connection establishment in spite of their lack of mobility. By building assumptions and services into the network itself, the system as a whole is made less flexible. When conditions change and assumptions fail to hold, the rigidity of cellular data systems causes them to break.

## 6 Constructing Robust Cellular Data Networks

Addressing the specific attacks detailed in this paper may be realistic in the short term. Optimized paging techniques [9, 25] may help to reduce search time and its resulting delay. As was done with the SMS attacks [47], techniques from queue and resource management could be used to mitigate blocking on the RACH. The move to 3G and a significantly larger pool of identifiers would re-

duce the practical likelihood of virtual resource exhaustion. While such methods would indeed mitigate many of the example vulnerabilities discussed in this work, a strategy for building robust cellular data systems based on constant patching would ultimately fail. All of the above solutions merely treat the symptoms of a larger problem. Accordingly, as long as there is a disconnect between the ways in which data is delivered in cellular and traditional data systems, exploitable mechanisms will exist. Such mechanisms need not be limited to the wireless portion of the network; rather, any component of the core network involved in establishing a session will be vulnerable.

The larger issue discussed in this paper, that of vulnerability caused by the exchange of traffic across two incompatible networks, will not be easily solved. Genuinely addressing this problem will require notable changes to the interaction between cellular data networks and end devices. Once such technique might require a significant increase of location awareness on the side of the network. Between the generation of paging lists and bandwidth used in multiple sectors, significant processing resources and time are spent finding a device each time a connection establishment occurs. Instead of knowing that a device is serviced by a potentially large set of base stations, an improved system might require location update information from a device each time it moves between sectors. Used in concert with much shorter sleep cycles, such an improvement to location knowledge may make the elimination of paging possible. This approach, however, would have a serious impact on resources in both end devices and the network. From the user perspective, increased monitoring and interaction with the network would negatively impact battery life. In the case of the latter, the overhead needed to process such an increase in messaging would also affect network performance. A more radical approach would be to replace cellular data services with a new high-bandwidth wireless protocol. Instead of necessarily sharing bandwidth and timeslotting schemes with voice communications, this new protocol would be assigned to a separate portion of the spectrum. In so doing, designers of the new data system would not be constrained by any of the rigidity forced upon current cellular data networks. In addition to technical tradeoffs, this solution would also need to deal with the complexities involved in spectrum allocation - reducing its viability for the foreseeable future.

These solutions are not an endorsement of any technology or architecture over another. Instead, they are simply the product of an observation of the impact on availability caused by interconnecting diametrically opposed methods of system design. Being beholden to a specific architecture and failing to understand the prob-

lems caused by linking such networks are in fact the causes of the rigidity seen in this system. It is highly unlikely that similar thinking will correct the problem.

## 7 Related Work

Representing perhaps the oldest functioning digital systems, telecommunications networks have evolved significantly since their inception over 100 years ago. While the nature of these systems themselves has transformed from manually configured and static to automated and mobile, many consumer behaviors have remained largely unchanged. Specifically, the frequency and duration of user calls have become largely predictable behaviors. System designers have used these anticipated conditions to optimize resource allocation throughout their networks. The degree to which telecommunications networks are tailored to such behavior quickly becomes obvious in the presence of unexpected changes to network usage. For example, the explosion in use of dial-up modems in the early 1990s caused widespread congestion because users were remaining connected for longer than expected time periods. Temporary fluctuations or surges, such as those seen minutes after the attacks on September 11th 2001, often render telecommunications networks unusable [35]. Such systems do not gracefully degrade under increased traffic volumes; rather, they often cease to provide service to the vast number of subscribers.

Recognizing this, our previous work focused on the ability to recreate the consequences of such high-traffic denial of service events through the use of low-bandwidth attacks. Using targeted loads of text messages, we were able to demonstrate the ability to deny voice and SMS service to major metropolitan areas with the bandwidth available to a cable modem [16]. We later characterized these attacks through simulation and measurement and discussed the tradeoffs inherent to a number of mitigation strategies [47]. Serror et al. [44] offered additional insight by exploring attacks on call paging channels. Ricciato [39] provided a general discussion of the potential to flood data channels in next generation networks with traffic generated by Internet-based pathogens. Raccic [36] and Mulliner [32] then examined attacks on MMS. While by no means the only methods of causing service outages, these attacks are the first to address the potential for denial of service made possible by the connection between cellular networks and the Internet.

Denial of service attacks have been studied in a variety of other contexts. Websites ranging from DNS roots [17], search engines [40] and software vendors [19] to online casinos [10] and news services [41] have all been temporarily disabled by overwhelming volumes of

traffic. Real-world processes and resources connected to the Internet, including banking networks, emergency services [30] and even postal delivery [13] have also been subjected to such attacks. In response, significant work has been undertaken to classify [29] and alleviate [22–24, 43, 46, 49–52] such problems. Unfortunately, none of these solutions have been widely deployed.

The debate over which network architecture is more resilient against such problems has raged for nearly 30 years. Advocates of the “smart” network, which is embodied by centralized control and decision-making, argue that this architecture provides the ability to prevent such overloading from occurring [31]. Supporters of “dumb” network architectures, which are built around the end-to-end principle [11, 12, 38, 42], contend that placing such control in the network itself dampens the ability to perform its intended task - routing packets. While both approaches have their tradeoffs, the discussion of the consequences of connecting systems that deal with transferring information in fundamentally different ways has not been addressed from the perspective of security.

## 8 Conclusion

Efforts to address recently discovered vulnerabilities in cellular networks have focused on treating symptoms instead of the disease. Attempts to solve individual exploits have been largely ad-hoc and, in their efforts to mitigate specific problems, create significant additional complexity and vulnerabilities in these systems. Without an understanding of why such attacks are happening, this cycle of vulnerability discovery and patching will continue indefinitely. The problems presented in this and other papers are artifacts of a larger architectural mismatch. Specifically, in spite of a concerted effort to support packet-switched traffic, cellular data networks are still, at their essence, circuit-switched systems. Because of this inflexibility, any mechanism responsible for connection establishment in these networks is vulnerable to a low-bandwidth denial of service attack.

We arrive at this conclusion by making the following contributions:

- Although conventional wisdom suggests that increased bandwidth provides robustness against such attacks, we use two new vulnerabilities to demonstrate that low bandwidth denial of service attacks can prevent legitimate access to cellular data services. In so doing...
- ... we demonstrate that a mismatch of bandwidth between cellular data networks and the Internet is not the cause of such attacks. Instead, they are the

result of the contrasting ways in which “smart” and “dumb” networks treat flows. From this...

- ...we show that in their uniform treatment of all flows, regardless of size or duration, cellular data networks exhibit design *rigidity*. By building significant assumptions about the behavior of traffic into the network itself, such systems are made brittle in the face of changing conditions.

Addressing these issues can therefore come from one of two approaches. In the first, methods of safely translating traffic between packet- and circuit-switched networks could be developed. Alternatively, such networks could be redesigned to truly support packet-switched mechanisms. By genuinely separating voice and data, not only in the spectrum they occupy but also in the techniques through which they are delivered, robust cellular data networks could be constructed. In the absence of such changes, cellular networks will continue to remain vulnerable to low-bandwidth exploits.

## Acknowledgments

This work was supported in part by Raytheon through a Wireless IR&D contract. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of Raytheon.

We would also like to thank Kevin Butler, William Enck, Joshua Schiffman and our anonymous reviewers for their invaluable comments.

## Notes

<sup>1</sup>We use the GSM architecture to provide specific details in our explanation. Similar mechanisms exist in other cellular networks.

<sup>2</sup>Enhanced Data rates for GSM Evolution (EDGE) is largely equivalent to GPRS. The most significant difference is the use of a new wireless modulation technique known as 8-phase shift keying (8PSK), which allows higher data rates.

<sup>3</sup>Note the subtle difference in naming. PDTCs are virtual channels that are run on top of physical PDCHs.

<sup>4</sup>This timer is referred to in the specifications as T3169 [2]. It is actually started when the counter N3101, which indicates the number of radio blocks that have passed since the last exchange with the targeted device occurred, reaches its maximum value. Our description above is meant to simplify the exact mechanisms for the reader without loss of precision.

<sup>5</sup>We consider connection establishment in terms of individual flows. Initial access to almost every network has a cost (authentication, etc). This startup cost, however, is amortized in both settings.

<sup>6</sup>At the time of this writing, Cingular Wireless had not yet been renamed AT&T.

<sup>7</sup>The voice network equivalent of the PRACH is employed due to the observed presence of dual-use control channels.

## References

- [1] 3G Newsroom. High speed mobile data driving uptake of PC cards. [http://www.3gnewsroom.com/3g\\_news/mar\\_06/news\\_6855.shtml](http://www.3gnewsroom.com/3g_news/mar_06/news_6855.shtml), 2006.
- [2] 3rd Generation Partnership Project. General Packet Radio Service (GPRS); Mobile Station (MS) - Base Station System (BSS) interface; Radio Link Control/Medium Access Control (RLC/MAC) protocol. Technical Report 3GPP TS 44.060 v7.6.0.
- [3] 3rd Generation Partnership Project. General Packet Radio Service (GPRS); Overall description of GPRS radio interface; Stage 2. Technical Report 3GPP TS 03.64 v8.12.0.
- [4] 3rd Generation Partnership Project. GSM/EDGE Radio Access Network; General Packet Radio Service (GPRS); Overall description of the GPRS radio interface; Stage 2. Technical Report 3GPP TS 43.064 v7.2.0.
- [5] 3rd Generation Partnership Project. Physical layer on the radio path; General description. Technical Report 3GPP TS 04.18 v8.26.0.
- [6] 3rd Generation Partnership Project. Technical realization of the Short Message Service (SMS). Technical Report 3GPP TS 03.40 v7.5.0.
- [7] 3rd Generation Partnership Project. Technical Specification Group GSM/EDGE Radio Access Network; Multiplexing and multiple access on the radio path. Technical Report 3GPP TS 05.02 v8.11.0.
- [8] 3rd Generation Partnership Project. Technical Specification Group Radio Access Network; Medium Access Control (MAC) protocol specification (Release 7). Technical Report 3GPP TS 25.321 v7.2.0.
- [9] A. Abutaleb and V. O. Li. Paging strategy optimization in personal communication systems. *Wireless Networks*, 3(3):195–204, 1997.
- [10] S. Berinato. Online Extortion – How a Bookmaker and a Whiz Kid Took On an Extortionist and Won. *CSO Online*, May 2005.
- [11] S. Bhattacharjee, K. Calvert, and E. Zegura. Active Networking and the End-to-End Argument. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 1997.
- [12] M. Blumenthal and D. Clark. Rethinking the design of the Internet: the end-to-end arguments vs. the brave new world. *ACM Transactions on Internet Technology (TOIT)*, 1(1):70–109, 2001.
- [13] S. Byers, A. Rubin, and D. Kormann. Defending Against an Internet-based Attack on the Physical World. *ACM Transactions on Internet Technology (TOIT)*, 4(3):239–254, August 2004.
- [14] Cingular Wireless. Cingular Wireless. <http://www.cingular.com/>, 2007.
- [15] D. Clark, J. Wroslawski, K. Sollins, and R. Braden. Tussle in Cyberspace: Defining Tomorrow's Internet. In *Proceedings of ACM SIGCOMM*, 2002.
- [16] W. Enck, P. Traynor, P. McDaniel, and T. F. La Porta. Exploiting Open Functionality in SMS-Capable Cellular Networks. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, November 2005.
- [17] R. Farrow. DNS Root Servers: Protecting the Internet. *Network Magazine*, 2003.
- [18] M. Grenville. Stats & Research: 3GSM Visitors Low Users Of Mobile Data. <http://www.160characters.org/news.php?action=view&nid=1950>, 2006.
- [19] C. Haney. NAI is latest DoS victim. [http://security.itworld.com/4339/NWW116617-02-05-2001/page\\_1.html](http://security.itworld.com/4339/NWW116617-02-05-2001/page_1.html), February 5 2001.
- [20] J. Hedden. Math::Random::MT::Auto - Auto-seeded Mersenne Twister PRNGs. <http://search.cpan.org/~jdhedden/Math-Random-MT-Auto-5.01/lib/Math/Random/MT/Auto.pm>. Version 5.01.
- [21] Hewlett-Packard. HP to Drive Mobile Connectivity Around the Globe with Vodafone. <http://www.hp.com/hpinfo/newsroom/press/2006/060706b.html>, 2006.
- [22] J. Ioannidis and S. Bellovin. Implementing Push-back: Router-Based Defense Against DDoS Attacks. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, February 2002.
- [23] A. Juels and J. G. Brainard. Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 1999.

- [24] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proceedings of ACM SIGCOMM*, 2002.
- [25] B. Krishnamachari, R.-H. Gau, S. B. Wicker, and Z. J. Haas. Optimal sequential paging in cellular wireless networks. *Wireless Networks*, 10(2):121–131, 2004.
- [26] C. Lepschy, G. Minerva, D. Minervini, and F. Pascali. GSM-GPRS radio access dimensioning. In *IEEE Technology Conference (VTC Fall)*, 2001.
- [27] C. Luders and R. Haferbeck. The Performance of the GSM Random Access Procedure. In *Vehicular Technology Conference (VTC)*, pages 1165–1169, June 1994.
- [28] K. Maney. Surge in text messaging makes cell operators :-). [http://www.usatoday.com/money/2005-07-27-text-messaging\\_x.htm](http://www.usatoday.com/money/2005-07-27-text-messaging_x.htm), July 27 2005.
- [29] J. Mirkovic and P. Reiher. A Taxonomy of DDoS Attacks and DDoS Defense Mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004.
- [30] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4), July 2003.
- [31] T. Moors. A critical review of 'End-to-end arguments in system design'. In *Proceedings of the IEEE International Conference on Communications (ICC)*, 2002.
- [32] C. Mulliner and G. Vigna. Vulnerability Analysis of MMS User Agents. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2006.
- [33] R. Mullner, C. F. Ball, K. Ivanov, and H. Winkler. Advanced quality of service strategies for GERAN mobile radio networks. In *IEEE Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2004.
- [34] J. Nagle. RFC 896 - Congestion Control in IP/TCP Internetworks. <http://www.ietf.org/rfc/rfc896.txt>, 1984.
- [35] National Communications System. SMS over SS7. Technical Report Technical Information Bulletin 03-2 (NCS TIB 03-2), December 2003.
- [36] R. Racic, D. Ma, and H. Chen. Exploiting MMS Vulnerabilities to Stealthily Exhaust Mobile Phone's Battery. In *Proceedings of the IEEE International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2006.
- [37] M. Reardon. ThinkPads to support Cingular 3G technology. [http://news.com.com/ThinkPads+to+support+Cingular+3G+technology/2100-1034\\_3-6017968.html](http://news.com.com/ThinkPads+to+support+Cingular+3G+technology/2100-1034_3-6017968.html), 2006.
- [38] D. Reed, J. Saltzer, and D. Clark. Active Networking and End-To-End Arguments. *IEEE Network*, 12(3):67–71, May/June 1998.
- [39] F. Ricciato. Unwanted Traffic in 3G Networks. In *ACM SIGCOMM Computer Communication Review*, 2006.
- [40] M. Richtel. Yahoo Attributes a Lengthy Service Failure to an Attack. *The New York Times*, February 8 2000.
- [41] P. Roberts. Al-Jazeera Sites Hit With Denial-of-Service Attacks. *PCWorld Magazine*, March 26 2003.
- [42] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments In System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.
- [43] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *Proceedings of ACM SIGCOMM*, pages 295–306, October 2000.
- [44] J. Serror, H. Zang, and J. C. Bolot. Impact of paging channel overloads or attacks on a cellular network. In *Proceedings of the ACM Workshop on Wireless Security (WiSe)*, 2006.
- [45] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Usenix Security Symposium*, pages 149–167, 2002.
- [46] A. Stavrou, A. Keromytis, J. Nieh, V. Misra, and D. Rubenstein. MOVE: An End-to-End Solution To Network Denial of Service. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2005.
- [47] P. Traynor, W. Enck, P. McDaniel, and T. La Porta. Mitigating Attacks on Open Functionality in SMS-Capable Cellular Networks. In *Proceedings of the Twelfth Annual ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2006.

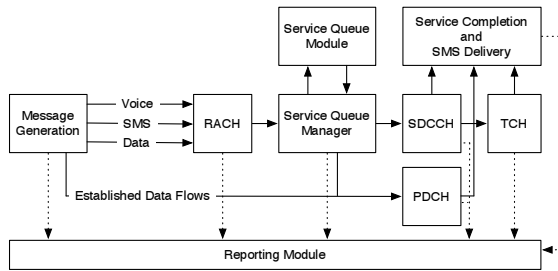


Figure 12: Simulator Architecture

- [48] United States Census Bureau. United States Census 2000. <http://www.census.gov/main/www/cen2000.html>, 2000.
- [49] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *Proceedings of Eurocrypt*, pages 294–311, 2003.
- [50] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. *Communications of the ACM*, 47(2):56–60, 2004.
- [51] J. Wang, X. Liu, and A. A. Chien. Empirical Study of Tolerating Denial-of-Service Attacks with a Proxy Network. In *Proceedings of the USENIX Security Symposium*, 2005.
- [52] B. Waters, A. Juels, J. Halderman, and E. Felten. New client puzzle outsourcing techniques for DoS resistance. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 246–256, 2004.

## Appendix

### Simulator Design

We extend the GSM simulator built in our previous work [47] to provide support for GPRS data service. In total, the project contains nearly 10,000 lines of code (an addition of approximately 2,000 lines) and supporting scripts. A high-level overview of the components is shown in Figure 12, where solid and broken lines indicate message and reporting flows, respectively. Traffic is created according to a Poisson random distribution through a Mersenne Twister Pseudo Random Number Generator [20], saved to a file and then loaded at runtime. The path taken by individual requests depends on the flow type. We focus on the data path as the behavior of SMS and voice messages were explained in the previous iteration of the simulator.

If the network has not currently dedicated resources to a flow on the arrival of a packet, it is passed to the

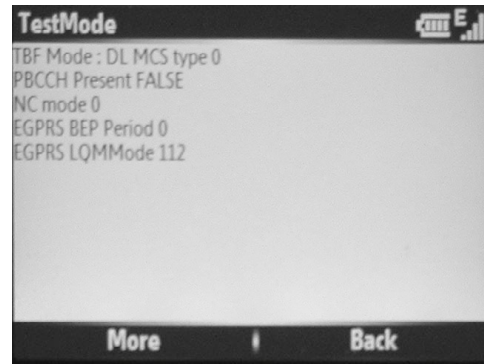


Figure 13: A Samsung Blackjack (SGH-i607) running in Field Test Mode provides operational data on the associated cellular network including channel configuration (shown here) and signal strength.

RACH module. This random access channel is implemented in strict accordance with 3GPP TS 04.18 [5] and is tunable via `max_retrans` and `tx_integer` values. Messages completing processing in the RACH are then delivered to the Service Queue Manager module, which in turn redirects data packets to the PDCH module. If a TFI is available, the packet is assigned the virtual resource, timers are set to five seconds and the packet is then delivered according to a FIFO ordering. The arrival of additional packets in a flow resets the timers to their default values to maintain resource control. When timers expire, the network reclaims a TFI for use in the delivery of other flows. Packets arriving at the Message Generation Manager as part of an active flow bypass the connection setup phases of the network and move directly to the PDCH module.

The accuracy of simulation was measured in two ways. The components used by voice and SMS were previously verified using a comparison of baseline simulation against calculated blocking and utilization rates. With 95% confidence, values fell within  $\pm 0.006$  (on a scale of 0.0 to 1.0) of the mean. The simple nature of the PDCH module allowed verification of correctness through baseline simulations and observation.

### Parameter Setting

When possible, we use settings found in currently deployed cellular data networks. However, such values are largely unpublished or unavailable to the general population. To find this information, we ran a Samsung Blackjack (SGH-i607) attached to the Cingular Wireless network<sup>6</sup> [14] in Field Test Mode. This mode of operation effectively turns a phone from a communications device to a network auditing platform. In addition to reporting the identification and signal strength read-

ings of nearby base stations, Field Test Mode provides network deployment information including channel allocation and layout. Accordingly, use of this mode of operation is typically restricted; however, access codes and device firmware upgrades are readily available online. As is shown in Figure 13 and of particular interest to properly modeling the behavior of real networks, the field `PBCCCH Present FALSE` indicates that voice and data control traffic use the same channels. This configuration, as previously discussed, is permitted by the standards [7] and effectively minimizes the amount of spectrum reserved for control information. Such a setting is believed to be common across the majority of provider networks. From these observations, the establishment of voice and data connections occurs over shared control channels in our simulations.

Other parameters are set using additional literature. For example, the `RACH` <sup>7</sup> is optimally set to reduce the probability of request blocking by allowing up to the maximum of seven retransmissions per request by the base station [27].

# Proximity Breeds Danger: Emerging Threats in Metro-area Wireless Networks

P. Akritidis\*, W.Y. Chin<sup>•</sup>, V.T. Lam<sup>†</sup>, S. Sidiroglou<sup>‡</sup>, K.G. Anagnostakis<sup>•</sup>

<sup>•</sup> *Systems and Security Department  
Institute for Infocomm Research (I<sup>2</sup>R), Singapore  
{kostas, wychin}@s3g.i2r.a-star.edu.sg*

<sup>‡</sup> *Computer Science Department  
Columbia University, USA  
stelios@cs.columbia.edu*

<sup>\*</sup> *Computer Laboratory  
Cambridge University, UK  
Periklis.Akritidis@cl.cam.ac.uk*

<sup>†</sup> *Dept. of Comp. Science and Engineering  
University of California San Diego, USA  
vtlam@cs.ucsd.edu*

## Abstract

*The growing popularity of wireless networks and mobile devices is starting to attract unwanted attention especially as potential targets for malicious activities reach critical mass. In this study, we try to quantify the threat from large-scale distributed attacks on wireless networks, and, more specifically, wifi networks in densely populated metropolitan areas. We focus on three likely attack scenarios: “wildfire” worms that can spread contagiously over and across wireless LANs, coordinated citywide phishing campaigns based on wireless spoofing, and rogue systems for compromising location privacy in a coordinated fashion. The first attack illustrates how dense wifi deployment may provide opportunities for attackers who want to quickly compromise large numbers of machines. The last two attacks illustrate how botnets can amplify wifi vulnerabilities, and how botnet power is amplified by wireless connectivity.*

*To quantify these threats, we rely on real-world data extracted from wifi maps of large metropolitan areas in the States and Singapore. Our results suggest that a carefully crafted wireless worm can infect up to 80% of all wifi connected hosts in some metropolitan areas within 20 minutes, and that an attacker can launch phishing attacks or build a tracking system to monitor the location of 10-50% of wireless users in these metropolitan areas with just 1,000 zombies under his control.*

## 1 Introduction

The last two decades of network security research have demonstrated that attackers are continuously evolving, exploring creative ways to exploit systems, and targeting new technologies and services as they emerge. Indeed, the widespread use of email brought spam and email-viruses; broadband connectivity was followed by the rise of rapid self-propagating worms; while the growing use of online personal services and electronic commerce resulted in sophisticated personal data theft attacks, including phishing. Such trends suggest that any technology that reaches some kind of critical mass *will* attract the attention of attackers.

At the same time, modern attacks such as worms, spam, and phishing exploit gaps in traditional threat models that usually revolve around preventing unauthorized access and information disclosure. The new threat landscape requires security researchers to consider a wider range of attacks: opportunistic attacks in addition to targeted ones; attacks coming not just from malicious users, but also from subverted (yet otherwise benign) hosts; coordinated/distributed attacks in addition to isolated, single-source methods; and attacks blending flaws across layers, rather than exploiting a single vulnerability. Some of the largest security lapses in the last decade are due to designers ignoring the complexity of the threat landscape.

The increasing penetration of wireless networking, and more specifically wifi, may soon reach critical mass, making it necessary to examine whether the current state of wireless security is adequate for fending off likely attacks. This paper discusses three types of threats

---

\*Part of this work was performed while P. Akritidis was visiting I<sup>2</sup>R under an industrial attachment programme

<sup>†</sup>This work was performed while V.T. Lam was working at I<sup>2</sup>R as a research engineer

<sup>‡</sup>Part of this work was performed while visiting I<sup>2</sup>R

- [20] A. Bittau, M. Handley, and J. Lackey. The final nail in wep's coffin. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 386–400, Washington, DC, USA, 2006. IEEE Computer Society.
- [21] D. P. Blinn, T. Henderson, and D. Kotz. Analysis of a Wi-Fi hotspot network. In *Proceedings of the International Workshop on Wireless Traffic Measurements and Modeling*, June 2005.
- [22] N. Borisov, I. Goldberg, and D. Wagner. Intercepting mobile communications: The insecurity of 802.11. In *Proceedings of ACM Mobicom, Rome, Italy*, July 2001.
- [23] S. Byers, L. F. Cranor, D. P. Kormann, and P. D. McDaniel. Searching for privacy: Design and implementation of a P2P-enabled search engine. In D. Martin and A. Serjantov, editors, *Privacy Enhancing Technologies*, volume 3424 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2004.
- [24] J. Cache and D. Maynor. Device drivers. Presentation at Blackhat USA 2006, August 2006.
- [25] R. G. Cole, N. Phamdo, M. A. Rajab, and A. Terzis. Requirements on worm mitigation technologies in MANETS. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 207–214, Washington, DC, USA, 2005. IEEE Computer Society.
- [26] T. Espiner. Does Wi-Fi security matter? [http://news.com.com/2100-1029\\_3-6088741.html](http://news.com.com/2100-1029_3-6088741.html).
- [27] F-Secure. Cabir worm description. <http://www.f-secure.com/v-descs/cabir.shtml>, June 2004.
- [28] F-Secure. Data Security Summary - January to June 2005. [http://www.f-secure.com/2005/1/data-security-summary-2005\\_1.pdf](http://www.f-secure.com/2005/1/data-security-summary-2005_1.pdf), 2005.
- [29] F-Secure. Inqtana.A worm information. [http://www.f-secure.com/v-descs/inqtana\\_a.shtml](http://www.f-secure.com/v-descs/inqtana_a.shtml), Feb. 2006.
- [30] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of RC4. In *Lecture Notes in Computer Science*, 2259:124, 2001.
- [31] B. Gedik and L. Liu. Location privacy in mobile systems: A personalized anonymization model. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 620–629, Washington, DC, USA, 2005. IEEE Computer Society.
- [32] M. Gruteser and D. Grunwald. Enhancing location privacy in wireless LAN through disposable interface identifiers: a quantitative analysis. *Mob. Netw. Appl.*, 10(3):315–325, 2005.
- [33] T. Henderson, D. Kotz, and I. Abyzov. The changing usage of a mature campus-wide wireless network. In *Proceedings of the Tenth Annual International Conference on Mobile Computing and Networking (MobiCom)*, Sept. 2004.
- [34] W.-J. Hsu and A. Helmy. On modeling user associations in wireless LAN traces on university campuses. In *Proceedings of the Second Workshop on Wireless Network Measurements (WinMee 2006)*, Apr. 2006.
- [35] Hu and Wang. Framework for location privacy in wireless networks. In *ACM SIGCOMM Asia Workshop*, 2005.
- [36] M. Hypponen. WLAN viruses, anyone? F-Secure weblog. <http://www.f-secure.com/weblog/archives/archive-082006.html>, August 2006.
- [37] M. Kim, D. Kotz, and S. Kim. Extracting a mobility model from real user traces. In *Proceedings of the 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, Apr. 2006.
- [38] B. Krebs. 2005 patch times for Firefox and Internet Explorer. Washington Post weblog. [http://blog.washingtonpost.com/securityfix/2006/02/2005\\_patch\\_times\\_for\\_firefox\\_a.html](http://blog.washingtonpost.com/securityfix/2006/02/2005_patch_times_for_firefox_a.html), Feb. 2006.
- [39] B. Krebs. Internet Explorer unsafe for 284 days in 2006. Washington Post weblog. [http://blog.washingtonpost.com/securityfix/2007/01/internet\\_explorer\\_unsafe\\_for\\_2.html](http://blog.washingtonpost.com/securityfix/2007/01/internet_explorer_unsafe_for_2.html), Jan. 2007.
- [40] N. Leavitt. Mobile Phones: The Next Frontier for Hackers? *IEEE Computer*, 38(4), April 2005.
- [41] C. Martel and V. Nguyen. Analyzing Kleinberg's (and other) small-world models. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 179–188, New York, NY, USA, 2004. ACM Press.
- [42] R. McMillan. Researchers hack wi-fi driver to breach laptop. InfoWorld. [http://www.infoworld.com/article/06/06/21/79536\\_HNwifibreach\\_1.html](http://www.infoworld.com/article/06/06/21/79536_HNwifibreach_1.html), June 2006. Accessed on September 15th, 2006.
- [43] J. Mickens and B. Noble. Modeling epidemic spreading in mobile environments. In *Proceedings of the ACM Workshop on Wireless Security*, pages 77–96, 2005.
- [44] S. Milgram. The small world problem. In *Psychology Today*, 1967.
- [45] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.
- [46] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of USENIX LISA*, November 1999. (software available from <http://www.snort.org/>).
- [47] T. S. Saponas, J. Lester, C. Hartung, and T. Kohno. Devices That Tell On You: The Nike+iPod Sport Kit. Technical report, Department of Computer Science and Engineering, University of Washington, 2007.
- [48] C. Shannon and D. Moore. The Spread of the Witty Worm. *IEEE Security & Privacy*, 2(4):46–50, July/August 2004.
- [49] S. Stamm, Z. Ramzan, and M. Jakobsson. Drive-By Pharming. Technical report, Indiana University, Dec. 2006.
- [50] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The Top Speed of Flash Worms. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 33–42, October 2004.
- [51] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the USENIX Security Symposium*, pages 149–167, August 2002.
- [52] J. Su, K. K. W. Chan, A. G. Miklas, K. Po, A. Akhavan, S. Saroiu, E. de Lara, and A. Goel. A preliminary investigation of worm infections in a bluetooth environment. In *WORM '06: Proceedings of the 4th ACM workshop on Recurring malcode*, pages 9–16, New York, NY, USA, 2006. ACM Press.
- [53] Symantec. Symantec Internet Security Threat Report. <http://www.symantec.com/enterprise/threatreport/index.jsp>, Sept. 2006.
- [54] Trend Micro. Botnet threats and solutions: Phishing - Whitepaper, Nov. 2006.
- [55] A. Tsow, M. Jakobsson, L. Yang, and S. Wetzel. Warkitting: the drive-by subversion of wireless home routers. *Anti-Phishing and Online Fraud, Part II Journal of Digital Forensic Practice*, 1(3), Nov. 2006.

that seem insufficiently addressed by existing technology and deployment techniques. The first threat is *wildfire worms*, a class of worms that spreads contagiously between hosts on neighboring APs. We show that such worms can spread to a large fraction of hosts in a dense urban setting, and that the propagation speed can be such that most existing defenses cannot react in a timely fashion. Worse, such worms can penetrate through networks protected by WEP and other security mechanisms. The second threat we discuss is large-scale spoofing attacks that can be used for massive phishing and spam campaigns. We show how an attacker can easily use a botnet by acquiring access to wifi-capable zombie hosts, and can use these zombies to target not just the local wireless LAN, but *any* LAN within range, greatly increasing his reach across heterogeneous networks. Last but not least, we discuss the use of Tracknets, city-wide wifi botnets for unauthorized tracking of user location and behavior.

All three types of attacks, illustrated in Figure 1, are specific to wireless networks, and are based on the premise of dense wifi network deployment in urban settings. While most of the underlying vulnerabilities have been widely known for years, the amplifying power of densely deployed wifi networking has a profound impact on both the feasibility and the magnitude of the threats, suggesting that their importance may have been grossly underestimated. For instance, the susceptibility of open wireless LANs to spoofing has been well documented, but the need to be in physical proximity to the victim may have deterred the wider use of this attack so far. The ability to launch such attacks remotely is much more attractive, and can be scaled up by the use of coordination and a botnet infrastructure.

As a result of underestimating these threats, no countermeasures are currently implemented. The mechanisms needed to thwart these attacks are in most cases either available but not actively used, or not available but relatively easy to implement. The fact that such mechanisms are not used is of particular concern. For instance, 802.11i security mechanisms have been available for several years, and would address a large part of the problems described, but unfortunately they are currently not used by enough users. Similarly, the encryption of MAC addresses would significantly increase the work-factor for Tracknets, but leaving the MAC addresses exposed was not deemed as a serious enough problem by the 802.11i group. Related to the worm problem, content-based filtering is widely used and intrusion prevention is a mature technology, yet to the best of our knowledge, it has not been employed in access point wireless-to-wireless forwarding. Raising awareness on the threats, using convincing, experimental evidence, is therefore at least as important as exploring and implementing possible defenses.

The main focus of this paper is in quantifying these threats, specifically in metro-area wireless networks. We rely primarily on publicly available maps of wireless access point locations, also known as wardriving maps, and attempt to derive estimates on the feasibility and effectiveness of the attacks using measurements and simulations. These estimates paint a grim picture on the exposure of current wireless networks to such attacks, and indicate that the risks are further increased as wireless penetration continues to grow as predicted.

We also explore possible remediation strategies, most of which we have implemented and tested experimentally. In some cases, the defenses we have considered are just a matter of engineering, such as retrofitting reactive worm defense hooks and filtering capabilities in wifi gear. In other cases, countering the threat required novel techniques, such as those for detecting and preventing different variants of the basic spoofing attack – several such variants were discovered while pondering about possible defenses, and how attackers might try to circumvent them.

While some of these techniques would become redundant if 802.11i is widely deployed, we cannot rest on the assumption that such deployment will happen anytime soon, particularly in light of usability concerns. For example, none of the recently announced municipal wireless initiatives that we are aware of employ any form of protection, most likely due to the current perception of the risks of open wireless as well as the cost of managing accounts and passwords for large number of users – in one instance, 100,000 users and around 9M annual visitors. Furthermore, the choice of running an open wireless network may not always be a matter of ignorance or complacency, but a conscious choice; for example, to provide network access to guests, backup connectivity to neighbors, etc [26]. Whether temporary or long-term, we believe that our supplementary defense techniques are useful for mitigating at least part of the threat.

## 2 Wildfire worms

The omnipresence and constantly improving capabilities of wireless mobile devices has attracted the regrettable attention of attackers, and in particular virus writers. The “Cabir” virus, which first appeared in 2004, was the first instance of mobile malware [27]. The virus exploited vulnerabilities in the Symbian OS and propagated through Bluetooth wireless connections. Experts predict the threat for smart phones and mobile devices is likely to increase significantly in the near future [40, 28].

Although such attacks *may* become prevalent in the years to come, in this paper we consider whether large-scale attacks are *already* feasible today on existing wireless infrastructure using current technology. In particular, we focus on worms that could spread entirely over

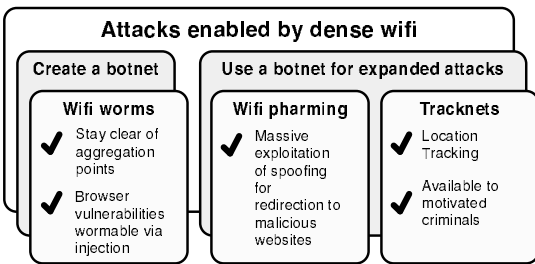


Figure 1: Dense wifi amplifies botnet threats.

802.11 wireless networks, even if such networks are completely heterogeneous. In this environment, the main concern is not necessarily the infection of mobile devices such as PDAs and cell phones, but the existing large population of laptops, desktops and other computers communicating over wifi. We consider worms that propagate entirely over wireless connections, trying to infect other computers tuned to the same access point (AP) and also other APs within range. A notable fraction of hosts in such an environment may also be mobile, and could therefore carry the infection from one AP to another. In densely populated metropolitan areas, it is conceivable that such a worm could infect a large fraction of wireless-connected hosts, especially considering pervasive vulnerabilities such as the ones exploited by Slammer [12], and recent browser vulnerabilities [13]. Such “client-side” vulnerabilities are of particular interest in a wifi setting, because unlike wired environments where a user needs to visit a malicious site to get exploited, it is often possible for an infected client to inject this kind of exploit via spoofing to any session between the target and a legitimate server. Considering the worst-case, a device driver exploit such as the recently discovered Intel driver attack [24, 36, 42] could carry the worm across platforms, and would even bypass VPN software which often blocks all local, wireless connections.

Although there has been considerable work in the literature on how to deal with large-scale attacks on traditional “wired” networks, there are at least three differences between wireless networks that require alternative solutions. First, wireless attacks can spread contagiously over wireless links based on proximity – similarly to real-world diseases – in contrast to the any-to-any communication possible over the Internet. This renders previous models and analyses of Internet-based worm propagation ineffectual as they cannot be directly mapped to wireless networks. Second, traffic in wireless networks is difficult to control using conventional methods, in lack of “hard” enforcement points such as firewalls between the communicating nodes. This is likely to significantly constrain the space for potential defenses. For instance,

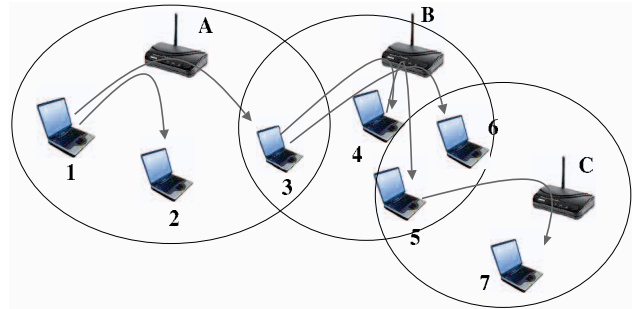


Figure 2: Simplified model of wildfire worm propagation.

if such a wireless worm were to be unleashed today, it would most likely go undetected by most, if not all, current attack detection infrastructures [17, 2, 3]. Finally, devices (*e.g.* handheld devices in the near future) in these environments are likely to be significantly more resource-constrained, at least in contrast to traditional desktop settings, and it is therefore more difficult and expensive to employ end-point security measures.

This paper is not the first to examine the threat of worms in wireless networks. Other researchers have made attempts at deriving contagion models in MANETs, examining viruses that spread according to user mobility, or measuring propagation dynamics in a campus network (these studies are discussed further in Section 6). Our paper is first to explore, in depth, the problem of wildfire worms and proximity propagation in densely populated areas. Specifically, we discuss the threat of worms that propagate entirely over wifi connections, and attempt to quantify the threat in terms of infection prevalence and infection timescales. Providing reliable estimates of potential infection prevalence is important for creating awareness on the severity of the threat, while the likely infection times are needed to guide the design of suitable countermeasures. Our analysis relies on simulated outbreaks of wifi worms driven by real-world data derived from wifi maps of large metropolitan areas around the world. Among other observations, our results suggest that a carefully crafted wildfire worm can infect all vulnerable wifi-connected computers in 80% of access points in some studied areas within 10-20 minutes – timescales at which traditional defenses may not be able to react in a timely fashion.

In this section, we describe the design and attack vectors of a wifi worm. The fundamental principle is that a wildfire worm relies on local, proximity-based propagation within shared medium broadcast environment such as WLAN.

## 2.1 Wifi worm propagation

Figure 2 illustrates the propagation dynamics of wildfire worms. Three access points A, B and C provide wire-

less coverage to end users, e.g. mobile nodes 1–7. They could represent, for example, WLANs deployed at adjacent buildings. Note that overlapping usually exists between adjacent access points for both residential networks (especially in densely populated cities) and corporate wireless networks (to allow for continuous connectivity and seamless mobility roaming).

Assume node 1 is the initial source of infection, *i.e.* it was infected previously at some other location before associating with access point A. Once activated, the worm analyzes WLAN A and probes all victims in the neighborhood; hence node 2 and node 3 eventually get infected. Note that node 3 is under coverage of both A and B. Normally node 3 picks and associates with only one access point, which is decided by certain criteria such as wifi signal-to-noise ratio. A worm-infected node, however, can gather a list of usable access points within reach and scan them for victims in the proximity. Effectively, the worm toggles association between usable WLANs to spread itself. Eventually all nodes in WLAN B and C are compromised through node 3 and nodes 5/6 respectively.

Nodes at coverage intersection of access points are “bridges” that help propagate the worm. These nodes can be thought of as “connectors” in the small-world phenomenon hypothesis [44, 41]. Contrary to the context of traditional Internet worms in which node 1 could probe and infect node 7 instantly, propagation dynamics of wildfire worms are similar to gradual and local diffuseness of disease. Therefore, a major advantage and difference of a wildfire worm over a regular Internet worm is that a wildfire worm can propagate entirely locally within each connectivity area, and thus evade firewalls and intrusion detection/prevention systems located at traditional enforcement points on the boundary between the local networks and the Internet.

Fertile ground for wildfire worms are wireless hotspot networks, which provide Internet access in public areas such as restaurants and airports, and private wireless networks of home users in residential areas. For example, Singapore government is realizing a “Digital Singapore” with wireless hotspots available at every street corner where people can log onto the Internet and receive emails on the move. Section 2.6.2 evaluates whether wifi penetration in metropolitan areas is sufficient for sustaining the spread of a wifi worm.

## 2.2 Mobility

Presently, the wireless node population consists mostly of laptops, and to a lesser extent of PDAs and smartphones (including wifi VoIP phones). The mobility patterns of wireless users can affect worm dynamics in three ways. First, mobility could compensate for sparse connectivity that may hinder wildfire-style propagation, as users carry the worm to networks previously unreach-

able by the worm. This is not restricted to just the places where the user turns on the laptop, as Laptops can also be programmed to wake up periodically as the user moves from one place to another. At the same time, user mobility also helps worm propagation into protected networks, whether they use WEP or more secure WPA/WPA2 protection, as the user will voluntarily (and perhaps even automatically) authenticate to those networks. Finally, the worm could create fake access points to lure and infect mobile users.

## 2.3 Open vs. Protected Access Points

There is a significant number of publicly available “open” access points; the rest are protected with Wired Equivalent Privacy (WEP) encryption or Wifi Protected Access (WPA). A worm can propagate over unprotected wireless networks in the way shown in Figure 2. Moreover, as a result of design and implementation flaws, WEP encryption is insecure. There is a handful of WEP attacks in the literature, e.g. weak IV attacks [30], keystream re-use [15, 22] and more recently fragmentation attacks [20]. These attacks are not just of theoretical value; they have been implemented into many practical and efficient WEP cracking tools freely available on the Internet. Wepcrack [8] did a performance comparison on some of such tools. Among them, Aircrack [1] is particularly powerful with a high success rate and relatively low cracking time that could vary between 5 seconds to 1 minute. However Aircrack needs to spend considerable time to sniff and capture sufficient wireless packets before cracking attempt. For example, after analyzing wireless usage statistics at a university campus [7], we determine that it may take 1-2 hours on average to successfully crack WEP encryption. Instead of passively sniffing packets, the worm could also employ active attacks e.g., discovering the encrypted version of a plaintext packet [8]. As for WPA, while not inherently weak, it is susceptible to brute-force attacks if used with a weak password in the most common WPA/PSK configuration. Given the apparent susceptibility of the currently available protection mechanisms, it seems likely that worms would consider carrying the additional payload of including cracking tools.

## 2.4 Infection process

In the design of a wildfire worm, we note that there are two possible ways to exploit vulnerabilities. The first approach, known as the “push method”, is to directly probe for an exploitable service and inject code to that service on clients just as traditional worms (e.g. DCOM RPC vulnerability on port 135 for Blaster worm). With the second approach, dubbed “pull method”, instead of relying on a service vulnerability, the attacker exploits vulnerabilities, such as browser vulnerabilities by per-

forming a man-in-the-middle attack. For example, the infected node can listen on the wifi and wait for the victim to make a DNS request, spoof the response pointing to itself (or some other, unused address), pretend it is the web-server and respond with pages that include exploits such as the WMF exploit [13] or other exploits for IE and Mozilla that attempt to execute malicious code. ARP spoofing and TCP injection attacks may be used as well. We note that the distinction between worm and virus is blurred in this case, as propagation may require some form of user interaction, yet the attack is piggybacked on communication to a third party, rather than between infected and targeted host. The broadcast nature of most wireless setups makes “pull” attacks attractive for wildfire worms as they can be exploited at a scale that was never possible for Internet worms.

## 2.5 Proof-of-concept implementation

We have implemented a proof-of-concept wildfire worm for both Windows XP and Windows Vista. This worm, dubbed Wildfire/A, has been submitted to security vendors for testing. The implementation of this worm was surprisingly straight-forward given the plethora of tools publicly available.

The WLAN API available for both Windows-Vista and -XP facilitates the process of managing AP association and scanning. Through this API, the worm is able to actively scan for open “visible” APs and, in turn, associate with them. Once associated with an AP, the worm scans the local subnet for vulnerable machines. For this particular proof-of-concept implementation we only considered push exploits, namely, the chunked-encoding vulnerability found in the Apache Web server 1.22. The worm payload is packaged as a self-extracting archive that contains the libraries required by the WLAN API as well as a copy of the actual worm. We have confirmed that the worm operates as expected in a small scale experiment with 4 APs and 15 vulnerable hosts.

## 2.6 Analysis

As with all worms, wildfire worms need to exploit a vulnerability to infect end-hosts. Unlike Internet worms that can effectively spread even if the vulnerable population is very small [48], wifi worms depend on the vulnerability being widespread. This raises two questions: what critical mass does a wildfire worm require to be effective, and whether there are indeed such pervasive vulnerabilities.

### 2.6.1 Vulnerabilities

To determine whether there is a significant number of pervasive vulnerabilities, we analyze vulnerability data from a variety of sources, including NVD [4], Securityfocus [6], and other independent sources. We focus on remotely exploitable vulnerabilities in the default in-

stallation of Windows XP Service Pack 2, between August 2004 (the Windows XP SP2 release date) and January 2007. We classify vulnerabilities based on whether they can be triggered through direct injection (“push” exploits) or through spoofing attacks as discussed in the previous section (“pull” exploits). Starting from basic information available through the NVD database, we verify the vulnerability information and derive further details such as exploit availability, exploitation technique, disclosure date, and patch dates primarily from Securityfocus archives but also other independent sources.

For all the qualifying vulnerabilities, we attempt to get a rough estimate of the *vulnerability window*: the amount of time the vulnerability was known and not patched in the majority of hosts. Unfortunately, publicly-available information does not always give us an accurate timeline of exploitation time vs. disclosure time, and we therefore have to make certain assumptions. In particular, we optimistically assume that by the time a vendor (in this case, Microsoft) releases an update, all hosts in the network are instantly updated and patched. In most (but not all) cases, the vulnerability is disclosed by the vendor only when the update is available. As such, it is not always possible to determine exactly when the vulnerability became known and to consider this as the start of the vulnerability window.

In lack of more accurate data, we assume that the vulnerability window starts two week before the update is issued, as Microsoft only posts updates every second Tuesday of each month. This is corroborated by Symantec which reported an average period of 13 days for the first half of 2006 between disclosure date of a vulnerability and the release date of an associated patch by Microsoft [53].

The results indicate significant exposure to vulnerabilities in the default configuration over the last two years, accounting for more than 50% of all days in the total period. Vulnerabilities of “push” type, i.e., that affect services and don’t need user interaction, were active for 105 days (11.89%) while “pull” type, i.e., that need user-interaction of some-kind, were active for 428 days (48.47%). We believe this observation suggests a trend, in which server/services components seem to be relatively robust when compared to client components. This is especially alarming in the context of wifi worms, because they are particularly suited for exploiting such vulnerabilities, and their abundance may give them another evolutionary advantage over Internet worms. Overall, we have found that 60% of the listed vulnerabilities had public exploits available for 391 days (44.28%) during the time period.

Other analyses of vulnerability exposure for the years 2004–2006 published on the Internet paint an even dimmer picture for “pull” type attacks. For a total of 284

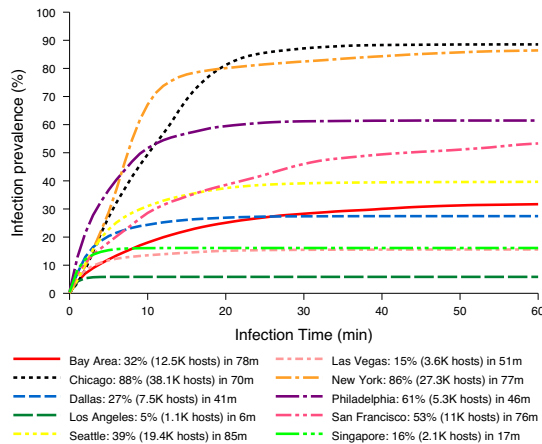


Figure 3: Spread of a wild-fire worm.

days (78%) in 2006, exploit code for known, unpatched critical flaws in pre-IE7 versions of the browser was publicly available on the Internet, and there were at least 98 days in which no software fixes from Microsoft were available to fix IE flaws that criminals were actively using to steal personal and financial data from users [39]. For at least 256 days (70%) in 2005, Internet Explorer contained unpatched vulnerabilities where the exploit method had been publicly disclosed but was not necessarily being used, and for at least 38 days in 2005, IE was vulnerable to unpatched critical security flaws that were being actively exploited [38]. A fully patched Internet Explorer installation was known to be unsafe for 98% of 2004, and for 200 days (54%) there was a worm or virus in the wild exploiting one of those unpatched vulnerabilities [11]. For Firefox, there were 56 days (15%) in 2004 where a publicly known remote-code execution had not yet been thwarted with a patch [11].

### 2.6.2 Worm simulation

To understand wildfire worm propagation, we simulate the outbreak of a worm in nine well-known US cities and Singapore. For this we relied on publicly available maps of Access Point locations from the *Wigle.net* [10] “wardriving” database, as well as empirically derived data for the city of Singapore. From these maps, we only consider open APs where the worm can spread without having to crack the encryption or the password.

Available war-driving maps chart APs but not connected hosts, so we had to populate them by randomly distributing hosts around APs. Based on our war-driving measurements and assuming a pervasive vulnerability, we distribute an average of 0.5 hosts per AP with Poisson distribution at an exponentially distributed distance of 10 m on average. We model effective AP range as omnidirectional with a radius of 90 m.

Finally, we do not consider the possibility of bypassing the AP to directly infect hosts within range using low level techniques because these depend on the available device driver and may not be widely available. We also ignore host mobility except that we assume the epidemic starts from 50 random locations to avoid artificially confining the worm to a sparse disconnected portion of the city.

The infection time for one hop is determined by four factors: scanning time, association time, IP acquisition time and transmission time. Based on our wildfire worm prototype, we assume a scanning and association time of about 1.5 seconds. We do not model DHCP interaction in our simulations as the worm can simply hijack an IP address. With an effective throughput of 14 mbps and 8 mbps for typical 802.11g and 802.11b networks respectively, the transmission speed is between 1 Mbytes and 1.7 Mbytes per second. Since the bandwidth will be shared among hosts, each host gets a transmission speed of a few hundreds kbytes/seconds. We assume a transmission speed of 100 kbytes/sec per host. For a worm size of 100K – which should be sufficient – the transmission time is about 1 second. A simulated worm-infected node infects its neighbours sequentially using these parameters.

Each simulation consists of 20 runs; for each run we start the infection from 50 different randomly selected hosts. We collect the mean values across runs of infection prevalence over time.

Figure 3 is a plot of infection prevalence over time for a “push” worm. Dense cities are infected very fast: 80% of New York and Chicago in less than 20 minutes. San Francisco and Philadelphia are infected fast as well: about 50% of San Francisco and Philadelphia are infected in 45 and 11 minutes respectively. A wild-fire worm does not spread significantly in Los Angeles and Las Vegas, but on a longer time scale a worm could still spread with the help of user mobility. The worm can spread fast as long as there are enough APs to maintain connectivity, but high density may even bog down the worm in some cases. In absolute numbers, we see that an attacker could quickly gain access to ten’s of thousands of hosts in most cities. The attacker could start simultaneous, independent epidemics in many cities using the Internet to infect a few seed hosts.

As for pull worms, we briefly summarize the simulation results here without a figure. Their simulated spread is limited compared to push worms – prevalence of pull attacks is limited to 60% in 3 hours for New York and Chicago, but they are potentially more dangerous, as they can take advantage of more vulnerabilities. They are slower because the infection time must include waiting for the victim to offer an opportunity for infection in the form of a DNS requests or TCP connection. On the

other hand, the worm can wait in parallel for any victim to become active. We use a very rough estimate of 10 minutes for waiting time to get an idea of the time scales involved, acknowledging that some machines may have no browsing activity at the time. The pull worm also requires higher density since we assume a shorter range of 60 m. Weaker antennas and increased interference typically weaken client transmission characteristics when compared with APs.

Overall, these time-scales suggest that automated defenses are crucial for defending against wildfire worms.

### 3 Large-scale Wifi Spoofing

One key property of open 802.11 networks is that they are built around a broadcast medium, where any wireless station can transmit wireless frames, and can listen to all other frames transmitted on the network. This is reminiscent of shared Ethernet segments of the 90's.

This property makes wireless LANs susceptible to spoofing and injection attacks, as discussed extensively in the context of wired Ethernet (but effectively disappeared with the emergence of switched Ethernet). The basic idea is that an attacker can monitor the communication between hosts on the wireless network, or between a host on the wireless network and an external party. If the communication is not properly encrypted, the attacker can elicit session state through eavesdropping, and if the communication is not authenticated, he can then *inject* frames to one session endpoint pretending to come from the other session endpoint.

Most protocols, such as DNS, DHCP and TCP are susceptible to this attack. In the case of DNS, the attacker can watch for outgoing DNS queries and inject responses pointing to a host under his control. For TCP the attack is similar – all the attacker needs to know is the current state of the connection in terms of sequence numbers. At connection setup, he may even completely take over the connection by injecting the proper SYN-ACK, resulting in the legitimate endpoint being out of sync. Injection is also possible at any point in the connection as long as the attacker can time injection attempts to properly deliver TCP segments to the victim network stack. The DHCP protocol can be spoofed to have a victim use an IP address and default gateway that gives the attacker full control over all of his traffic. However, it may be less attractive than DNS and TCP spoofing as the attacker has to wait for the victim to refresh his DHCP lease, or else attack only hosts that have connected after the attacker has obtained access to the wifi network.

While in the 90's such attacks were seen as enablers for unauthorized access, in today's threat landscape they are more likely to be used for "modern" attacks such as phishing, spam and exploit injection. In the previous section we briefly discussed how injection can be used to

propagate a worm through client-side vulnerabilities. In this section we focus on spoofing primarily for the case of launching phishing attacks, and discuss ways to detect and prevent them. DNS spoofing is highly attractive for phishing as, for example, the attacker may set up a mock banking website that would relay manipulated requests to the real site in a man-in-the-middle fashion. We note that in this case, two-factor authentication cannot help. Similarly, TCP injection can be used to insert redirection instructions, advertisements, or spam to otherwise legitimate Web pages. Sophisticated attacks can even subvert user's services, such as using a victim gmail account, etc.

The use of such techniques in wifi for phishing has been documented previously. The so-called "parking lot attack" involves the attacker being in physical proximity to the target network. While this attack may be interesting by itself, we are not aware of any extensive use of this technique. One main disadvantage is that the physical proximity constraint increases the risk to the attacker, especially in environments with pervasive CCTV coverage that can be used for forensics. In the context of this paper we explore how proximity enables remotely controlled bots to be used for such activities. In this case, the attacker can acquire access to a wifi-enabled host located in a wifi-rich location. In contrast to traditional Trojans, the attacker need not try to elicit information from the owner of the actual machine that is being exploited. Rather, the attacker may perform spoofing on *any* wireless network within range from the host under his control using channel hopping and/or temporary association for the duration of the attack. The dense use of wifi in metropolitan areas makes this model quite attractive, as it may significantly *amplify* the attacker's capabilities.

#### 3.1 Analysis

To determine the effectiveness of spoofing attacks in terms of scale we rely on the same publicly-available wifi maps used for analyzing wildfire worms. We attempt to get a rough estimate of the number of access points that hosts on the map can connect to. As we only have access point locations, we add hypothetical hosts within range from each access point. We distribute 1 host per AP and assume a communication range of 60m.

We compute the number of neighboring APs for each host, that is, all APs within range excluding the AP it is directly connected to. We consider only "open" APs that do not use any wireless security protocol, even though the attacker may well be able to crack into WEP-enabled networks using well-known attacks and tools.

The results for our analysis on 10 different metro areas are shown in Figure 4. We see that in half of the cities 90-99% of all hosts can connect to at least one more neighboring AP; 20-50% of hosts can connect to at least 10

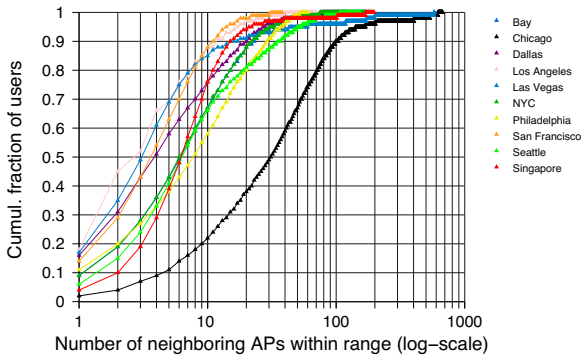


Figure 4: Number of WLAN networks observable from random hosts in metro areas (range 60 m).

additional APs; and a small but non-negligible number of hosts, as high as 10% in Chicago are within range of more than 100 APs. Unsurprisingly, the results are worse for Chicago, which seems very densely populated, and less so for relatively sparse areas.

Overall, the results confirm our fear that controlling wifi-enabled hosts in densely populated areas can be highly attractive to attackers.

#### 4 Wifi tracknets

The proliferation of city-wide wifi networks has already raised serious concern over privacy implications. Privacy advocates fear that wifi networks can be used to record location information for the operating ISPs, their partners, and possibly law enforcement, raising concerns that wifi can be used to track general user behavior in a "Big Brother" fashion.

However worrying this scenario might appear, it can be classified as a mere nuisance when compared with the possibility of *anyone* being able to remotely set up a tracking system, without even having to set up physical infrastructure. Such systems, which could be termed as *Tracknets* can be deployed using a reasonably sized botnet, providing a user-tracking mechanism that can operate across wireless network boundaries. Criminal gangs are known to operate marketplaces for bots, sometimes with specific features such as high bandwidth and CPU power, priced between \$1 and \$40 per compromised PC according to security experts who have monitored IRC chat room exchanges [54]. It is conceivable that attributes such as wifi connectivity, and location within a metro-area could be added to the list of features to facilitate attacks such as those described here.

Such a botnet can then track location information [16], possibly coupled with user-profiles that can span across heterogeneous wireless LANs. The location of the zombies comprising the bot can be inferred from the ESSID

attacks	defenses				
	Ingress Filtering	Packet Rewriting	802.11 Spoofing Detection	Inconsist. Duplicate Detection	Whisper Attack Detection
Baseline Spoofing	✓✓✓		✓✓	✓✓	✓
External Collaborator		✓✓✓		✓✓	✓
802.11-level Spoofing			✓✓	✓✓	
Whisper Attack					✓

Figure 5: Spoofing defense space.

of their AP using public wifi maps. (In fact, this service is already provided by companies such as Navizon and Skyhook.) The number of users that can be tracked using Tracknets and its coverage are commensurate with the size of the botnet population and the amplifying effect of proximity, similar to the spoofing threat discussed in the previous section.

Several services can leak significant amounts of privacy-sensitive information. This information can, in turn, be used for targeted Phishing and spam attacks, blackmail, and for pre-attack reconnaissance such as building hit-lists. In addition to high-information-leak vectors, several techniques can provide personal information at a lower granularity that might not be able to distinctly identify individual users but can be used to classify sets of users according to broader set of criteria such as OS version, wireless driver information and general browsing behaviour. In this section we briefly examine some of the most obvious tracking vectors. Our investigation is far from exhaustive and only scratches the surface of possible ways that users could be tagged and tracked. Nevertheless, the vectors we discuss show *at least* one set of techniques that seem threatening enough by themselves, and may be representative of other approaches.

**MAC address** The obvious way to track users across heterogeneous WLANs is to use the MAC address as unique identifier. Trackers can use this information to correlate any other behavioral information to a MAC address to easily create profiles. Fortunately, although MAC addresses are permanent by design, there exist a number of mechanisms that allow users to change the identifier. Gruteser et al [32] introduce the idea of short-lived disposable MAC addresses as a technique for the reduction of the effectiveness of location tracking. However, randomizing MAC addresses often leads to problems. For example, several ISPs use MAC addresses

to map IP addresses. Also, some software licenses are bound to a specific MAC address. Furthermore, even in the presence of such techniques, user profiling can still effectively track users in dense urban environments. In our system, we use MAC addresses as temporary identifiers for correlating information that will be used to create user profiles as described below.

**Live bookmarks – RSS** Live bookmarking is a new popular method for displaying web feeds as bookmarks. Its popularity surged when it was introduced in Mozilla Firefox 1.0 back in 2004 and can now be found in several other popular web browsers such as Apple's Safari and Internet Explorer 7. Live bookmarks subscribe to user-defined RSS feeds and are periodically updated so as to display the latest articles. The ability to customize feeds along with the inherent periodicity of the updates make Live Bookmarks susceptible to eavesdropper profiling. In particular, as users subscribe to more RSS feeds they inadvertently create distinct profiles that can be used to track them. Given the wide range of tools available for parsing RSS feeds, it is trivial for a tracker to parse the feeds so as to extract user personalization in addition to RSS subscription information. Worse, by using traffic analysis to identify such communications based on their periodicity and creating a signature based on packet size distributions, an attacker could possibly track users over encrypted WLANs, however, we have not investigated this scenario further.

Tracknet bots would collect and parse all requests to RSS feeds. The information derived from the feed is then associated to an individual node. The node is temporarily identified by IP and MAC address for the current session. Any other information that is collected from the particular node is collected in a tracking tuple that correlates all other pertinent fields that aid in the identification of the node. In order to reduce the number of identification false positives we correlate the RSS fingerprint with the base station ESSID. Distinct fingerprints that appear at the same location (*e.g.* home or workplace) might point to a distinct identity with a higher level of confidence.

**Location tracking** Collaborating bots can use radio signal characteristics of WLANs to determine a user's location with relative accuracy using triangulation techniques. This information, in combination with other extracted personal information can lead to considerable privacy leaks. Specifically, bots can use this information to infer user behavior. For example, information on entertainment habits, political orientation, medical information can be potentially derived.

**Other services** Beyond the mechanisms described above, there are numerous other protocols and services that leak significant personal information. For example, numerous Instant Messaging (IM) system do not employ

encryption so all user identification information is available to eavesdroppers. Although this information might not be significant on its own, when it is correlated with other sensitive information, it can be used to construct a distinct user profile. Other systems that can be used to fingerprint user behavior are the mail servers that users connect to, information from other networking protocols such as NETBIOS and AppleTalk and even which VPN servers a user connects to.

The growing popularity of Google and other online service portals, has moved a number of user services to central aggregated locations where users can check their RSS feeds and email. Although this configuration changes the network fingerprint that is emitted by services it does not reduce the amount of information that is leaked. For example, the Google homepage includes links to personalized RSS feeds including the user's email address in plain text, which often points to a user's real identity, *e.g.*, john.doe@gmail.com. This information can be readily used to create very accurate user profiles since a tracker can intercept these unencrypted HTTP transfers.

Another serious vector of information leak is (to no surprise) the use of cookies. Cookies are used extensively as a mechanism for servers to identify users and track their access. The threat of Cookies to user privacy has received considerable attention in the literature [23]. In the context of tracknets, the exchange of Cookie information can be used to extract personalized user information based on both the contents of the Cookies and their transmission fingerprint. For example, Google, a company synonymous with Internet search uses cookies that expire in 2036. The cookie uses a 16-digit identifier to track user preferences and, inevitably, track user behavior. Given the popularity of the search engine, it is not unreasonable to assume that a large percentage of the user population will emit this identifier during its lifetime, adding another mechanism for user tracking.

The Dynamic Host Configuration Protocol (DHCP) is a ubiquitous protocol used for automating network configuration. Unfortunately, there is no privacy protection for DHCP messages, so an eavesdropper who can monitor the link between the DHCP server and requesting client can discover the information contained in this option. For example, the following snippet illustrates the kind of information that can be derived from a DHCP request. Information on the types of services and more importantly hostname information is made readily available to eavesdroppers.

```
Client IP: 10.50.16.205
Client Ethernet Address: 00:17:f2:40:61:65
Vendor-rfc1048:
DHCP:REQUEST
PR:SM+DG+NS+DN+NI+NITAG+SLP-DA+SLP-SCOPE+LDAP+T252
MSZ:1500
```

```
CID:[ether]00:17:f2:40:61:65
LT:7776000
HN:"alamak"
```

We collect and correlate the information derived from DHCP headers. In particular, we are interested in user-identifying information such as the user's hostname. This information might appear innocuous but is often linked to personal information such as the user's name or company information. Again, in this case we associate DHCP-derived information with the base station's ESSID.

#### 4.1 Experimental analysis

We determine how effective an attacker can be in tracking users using a botnet consisting of wifi-enabled hosts within a metropolitan area. For this purpose, we rely on the same wifi maps used for analyzing the worm and spoofing attacks. The effectiveness of a tracknet can be expressed in terms of *coverage*, that is, the fraction of wireless LANs that are within range from a given set of subverted nodes participating in the tracknet. The feasibility of a tracknet also relates to the number of subverted nodes that the attacker needs to obtain in order to achieve a certain level of coverage. As the attacker may have little control over which hosts to subvert (or buy access to) and where they are located, in each experiment we assume a random subset of hosts on the wifi map. As MAC addresses are exposed even when the network uses WEP or 802.11i encryption, we consider all access points regardless of whether they are open or protected – in other words, a tracker can monitor *any* network within range.

The results for 10 metro areas are shown in Figure 6. We observe that the fraction of subverted hosts needed to track users is relatively modest: with hosts on just 1% of all APs in a dense area, a tracknet can cover between 5% and 40% of all traffic. As expected, full coverage is not easy to achieve, but having trackers on around 7% can reach between 30% and 80% coverage. As with the worm and spoofing threats, the high density of Chicago and NYC make them particularly susceptible to this attack: less than 1,000 zombies are sufficient to cover 40% of the APs.

At the time of writing this paper, all MAC addresses are exposed, but it is worth investigating whether using disposable MAC addresses would help address this problem. As discussed previously, we are particularly concerned about other high-information-leak profiling techniques that could essentially offer uniquely identifying information equivalent to a MAC address. We focus on RSS feeds as one emerging source of leaks, and try to quantify the ability of an attacker to use this information for tracking purposes. For this purpose, we have obtained from an online service provider the set of RSS feeds that users are subscribed to, for around 100,000 users. The

size of the dataset is important as we seek to measure the *uniqueness* of each RSS profile. We therefore measure for each user, whether any other users have the same exact profile, in which case we say that we have a profile collision (which could make tracking information ambiguous and confusing to the attacker). As some users have empty or very small profiles, we expect more collisions there, and we therefore compute collision statistics for those users with at least a minimum number of feeds in their RSS set.

The results are presented in Figure 7. As expected for a minimum RSS set of zero, that is, no constraints, the fraction of users with colliding profiles is around 30% – most of them are users with an empty profile. Removing only those that have an empty profile, that is, focusing on a minimum set of one entry, the collision probability is 0.02 to 0.07, significantly lower and reasonable enough to allow a tracknet to identify a user with high confidence, especially given that this information can be correlated with other data. For users with more substantial RSS feeds, the collision probability is between 0.002 and 0.01, indicating highly unique profiles. The scaling behavior of collision statistics is of particular importance here: we see that collision probability increases with the number of RSS profiles in the dataset, yet the difference seems to be small between a database of 50K users and a database of 100K users. If a tracknet is supposed to cover a whole city, the number of profiles can be much larger than the set we considered here, but our results suggest that collision probability is unlikely to worsen significantly. Furthermore, when a user's RSS fingerprint is coupled with location information such as mobility patterns, this set can be reduced even further.

## 5 Defense strategy

The threat of wildfire worms and large-scale spoofing can be reduced significantly with the use of existing wireless security standards such as WPA/WPA2, with strong encryption and hard-to-guess passwords. Unfortunately, despite the wide availability of such techniques, users do not seem to employ them. Even if this is simply because there have been no large-scale attacks yet, the use of passwords hinders usability and robustness. It is likely that even if such measures are implemented, in many cases the passwords are not going to be strong enough to resist brute force attacks. As such, it seems worthwhile investigating alternative, reactive defenses specific to the attack vectors discussed so far. In the remainder of this section we discuss such defenses, as implemented in a prototype system for automated defense against wildfire worms and spoofing attacks based on the Linksys OpenWRT [5] router and optionally using an external controller and centralized threat analysis.

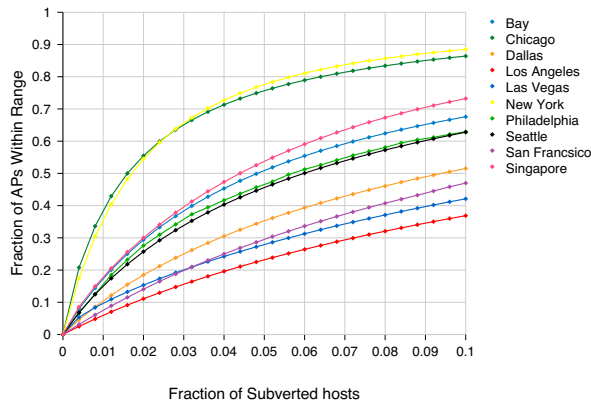


Figure 6: AP coverage of a given fraction of random bots in a metro area assuming a range of 60 m.

### 5.1 Wireless IPS

In our implementation, we have adapted the snort IPS [46] to run on OpenWRT. While previous implementations have used snort to filter traffic between the wireless network and the external ethernet connection, our implementation disables the normal low-level wireless-to-wireless forwarding and uses ebtables and IPtables to redirect traffic through userland where it can be processed by snort.

As APs typically have limited computing resources, it may not be possible to have a fully fledged IPS running on them. Increasing their capabilities may also be prohibitively expensive. There are at least two possible options to address this problem. The first option is to use only a subset of the signatures, most likely signatures for attacks against vulnerabilities that may not be universally patched yet. The second option is to implement the IPS functionality in a centralized wireless controller, and have the APs forward all local traffic for inspection before retransmitting to the wireless medium.

The main advantage of using a wireless controller is that it provides flexibility for devoting more resources to traffic inspection. It is also consistent with industry trends towards cheap, “dumb” access points managed by a wireless switch. However, none of the wireless switches we are aware of provide any filtering capabilities for internal WLAN traffic such as wildfire worms. In our case, the additional wireless-to-wireless IPS functionality is implemented as a standalone wireless controller. This functionality can be retrofitted into wireless controllers or implemented as part of a secondary controller – protocols for AP to controller communication are being standardized, and thus interoperability is likely to be achievable.

For zero-day attacks for which there are no signatures,

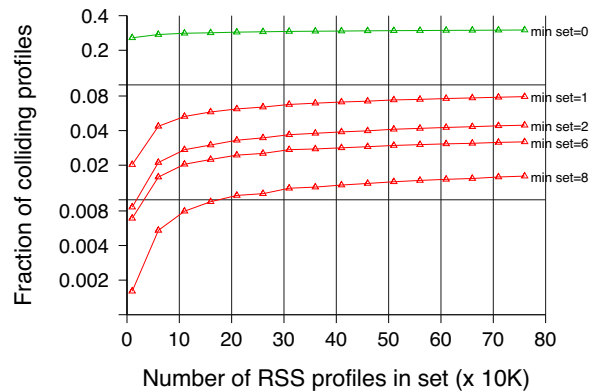


Figure 7: RSS set uniqueness.

we rely on honeypot feeds from access points back to an analysis center. There, we use the Argos system [45], which uses dynamic taint analysis to trap the execution of remotely-injected code, for detection coupled with our custom signature generation system. Our system collects packet trace samples corresponding to the exploitation attempts detected by Argos and then uses a heuristic for generating network-level attack signatures in the form of simple patterns. The heuristic tries to identify a substring that is sufficiently large, sufficiently frequent in the attack samples and sufficiently infrequent in benign traffic. This last part is important for addressing concerns about false positives as well as attempts to manipulate signature generation for denial-of-service purposes. Our implementation uses a novel inverse indexing scheme on previously collected packet traces. While in our test setup these traces are maintained centrally at the threat center, it is conceivable that such testing can be performed at each site independently. These signatures are then installed on the AP or the wireless controller as snort sensor rules.

Filtering of internal WLAN traffic assumes that the worm does not tamper with the wireless device driver and firmware. If such tampering is possible, the attacker may spoof access point transmissions directly – for which tools are publicly available [9], and bypass filtering mechanisms applied to traffic relayed over the AP. The AP can detect attempts to impersonate it as long as it can pick up the messages sent by the attacker, but this leads immediately to another attack scenario: the attacker can hide his emissions from the AP by tuning the wifi radio power or using directional antennas so that the spoofed packets can reach the victim, but cannot reach the AP (or external detection device). We refer to this as the *whisper* attack. The attack seems difficult to engineer, as it requires both the low-level driver/firmware

hacks of the basic 802.11 spoofing attack, as well as careful tuning of the radio. Unfortunately, newer chipsets provide improvements in power control, and it is likely that the attacker can easily find the “right” power setting to launch the attack by probing both the victim and the AP with different power settings, all controlled through the driver API. In some cases, the relative positions of AP, victim, and attacker may prevent this attack. In addition, not using an AP to relay frames limits the communication range. Using power control to evade the AP may limit the range even further, to the point where it may become impractical to perform whisper attacks in the context of the massive attacks we discussed.

## 5.2 Spoofing defense strategy and attack-defense co-evolution

Assuming WPA and VPN solutions comes with a considerable usability cost; we investigate lightweight alternatives. Interestingly, our exploration of defenses against spoofing attacks has revealed a small arms race. In particular, while developing defense techniques we discovered several new variations of the attack, each defeating one of our countermeasures. In this section, we discuss the attacks, the countermeasures and present results evaluating their effectiveness. These findings are summarized in Figure 5. We focus on DNS spoofing for simplicity, but in most cases the attacks and countermeasures are similar for other protocols.

### 5.2.1 Wireless ingress filtering defense

As discussed previously, the simplest form of DNS spoofing involves the attacker lurking for DNS requests to the target site, and then injecting a fake DNS response pointing to a site under the attacker’s control. It seems straightforward to defend against this attack through the use of *ingress filtering* at the AP. Ingress filtering ensures that all traffic broadcast by the AP on the wireless network is checked in terms of IP address and the interface on which it is received. That is, traffic originating from the wireless network should have IP addresses on the local wireless network. (Similarly, but less relevant here, traffic from the external network should not have an IP address on the internal network.) A DNS request is usually sent to a resolver outside the wireless LAN, and therefore the DNS response is expected from an external address. A spoofed response is trivial to detect, as it arrives on the AP from the wireless interface and has an external IP address.

### 5.2.2 External collaborator attack

A variation of the spoofing attack that circumvents ingress filtering involves the use of an external collaborator. In this variation of the attack, the attacker is again eavesdropping on the wireless LAN lurking for DNS requests, but instead of sending the spoofed response from

the wireless LAN, signals another host on the Internet to send a spoofed response to the victim. Being able to eavesdrop is crucial, as it allows the attacker to relay the needed DNS identifier and port number information to the remote collaborator.

There are two constraints for the attacker that make this attack more difficult. First, the remote collaborator needs to be able to send packets with the source IP spoofed. Unfortunately, a recent study [18] shows that spoofing is still possible on more than 30% of hosts due to the limited use of source filtering. Second, the remote collaborator needs to send the spoofed DNS response before the legitimate DNS response arrives. Thus, the attacker would need to locate a collaborator that is closer by in terms of round-trip times.

### 5.2.3 Packet rewriting defense

One way to defend against this attack is to rewrite packets as they flow through the AP to the outside world, mapping the DNS id and port number, TCP sequence numbers, etc., to a different space, then doing the corresponding inverse mapping on packets on the way back. The eavesdropper only knows the internal representation of those identifiers and cannot relay the necessary information to the external collaborator. Any spoofed response from the external collaborator will be transformed to have an identifier that will result in the response getting dropped by the victim, making the attack ineffective.

The mapping can be done using either a hash function, or a state table, and is robust as long as the mapping is unpredictable. In the case of hashing, we need to use a keyed hash, with the key being the destination IP address, to prevent the attacker from using a third-party DNS server to map out the key space. The choice between state table and hash function is not always clear, as it involves space-time tradeoffs. If the hardware provides cheap hashing, then it may be preferred. In our Linksys OpenWRT implementation the use of a state table was more efficient as hashing introduced a high per-packet cost that turned the technique into a bottleneck.

### 5.2.4 802.11-level spoofing attack

As discussed in the context of wifi IPS, a sophisticated attacker can circumvent the ingress filtering defense by violating the 802.11 protocol to transmit frames directly to the victim. The AP can detect this by monitoring for transmissions that it did not send. However, it cannot detect the whisper attack discussed earlier, where the attacker tunes the wifi radio power so that the spoofed packets can reach the victim, but cannot reach the AP (or external detection device).

When filtering fails, the next best option is to detect and forcibly abort the attack. We pursue this direction in

the next section.

### 5.2.5 Whisper attack detection

We have developed a set of defenses based on the detection of abnormal combinations of network events. For example, to detect the injection of a DNS reply, we use bookkeeping of request-reply pairs to flag excess, inconsistent replies. We also raise an alert when a host appears to retransmit requests after having received replies, so we can prevent a situation where the attacker keeps inserting a fake request, just before the legitimate reply for the previous request arrives, in order to maintain the request-response balance.

While there are no visible duplicate replies in case of a whisper attack, the AP may still detect the attack indirectly. A solution for HTTP is to extract from each HTTP connection the server hostname from the corresponding mandatory HTTP header and the server address from the IP header, and compare this pair against the hostname and IP pairs extracted from observed DNS replies. If a reply has been whispered, no DNS reply will match the HTTP header and the attack will be detected.

We have evaluated our technique for detecting whisper attacks against 41,426 DNS and 339,317 HTTP requests generated by 65 IP addresses over a period of a week. We obtained 18 alerts (6 unique web sites), all of them false, corresponding to a false positive rate of  $0.53 \times 10^{-4}$  of all HTTP requests. For the same trace we observed zero excess DNS replies. We further evaluated only our first technique for detecting excess DNS replies against 43,272,448 DNS requests obtained over a period of more than 1 month by instrumenting an enterprise network with about 400 users. We obtained 22 alerts, all of them false, corresponding to a false positive rate of  $0.5 \times 10^{-7}$ . Looking deeper into the alerts revealed a Content Delivery Network that is employing spoofing, probably for server selection.

Once an attack is detected, it has to be blocked. However, given two inconsistent DNS responses, the detector cannot directly distinguish which one is legitimate and which one is spoofed. Doing a secondary lookup is one option, but the wide use of load balancing, particularly for popular services, implies that the secondary lookup may not always agree with one of the two inconsistent responses. A more relaxed check against the network prefix is also unlikely to help in the general case, as server replicas may not be co-located. In lack of any other satisfactory solution, our current implementation blocks the victim and redirects him to a warning page, notifying him of a potential spoofing attack, and giving him the option to proceed (and re-issue the request) through temporary HTTP redirection.

## 6 Related work

With the growing popularity of mobile devices, malware targeting wireless environment have started to emerge [27, 29]. This new security challenge has recently gained some attention from the research community.

A study related to ours is the one by Tsow et al [55]. The authors suggest that attackers could drive around a city taking over vulnerable wireless home routers. Similar to our study, the threat is amplified by dense wifi deployment, as attackers can take over hosts at a higher rate. However, the attack depends on vulnerable access points, and requires the physical presence of the attacker for driving around to find vulnerable routers. The attacks we discuss in this paper can all be launched remotely, and therefore easier and less risky for the attacker.

Anderson *et al.* [14] analyzed the speed of worm contagion over campus-wide wireless networks. They developed a worm simulation using real data from Crowdad, *e.g.* user distribution, AP distribution and user mobility, to realistically study the dynamics of a mobile worm. However their results are constrained to dynamics of mobile worm at relatively small scale of a university campus with mobility as the major factor for worm spread. In contrast, our work has investigated big cities and metropolitan areas at much larger scale with wardriving data around. We have identified a much larger threat *e.g.* infection completion in the order of minutes whereas Anderson *et al.* [14] predict a few hours to infect just the campus. The main difference is that wildfire-like propagation—not just user mobility, is the key attack vector in our work. It is also unclear whether their defense proposals could be proven effective given recent major changes of wifi usage pattern.

Beyah *et al.* [19] discuss a worm that spreads by infecting users sharing the same hotspot. They use epidemic models to simulate its spread and find it can infect a million users worldwide over the course of a year. Again the main difference is that the simulated worm relies on user mobility, but we show using wardriving data that mere density is sufficient in metropolitan areas leading to much faster spread.

Su *et al.* [52] investigate worm infections in a bluetooth environment. They expect Bluetooth to outnumber wifi devices by a factor of 5 and predict large scale epidemics, but the short range of bluetooth again implies slower, mobility-based spread. Cole *et al.* [25] use epidemic models and simulations to discuss requirements for worm mitigation in tactical battlefield MANETS.

Stamm *et al.* [49] discuss remote attacks on routers that can be used for large-scale pharming and can also spread virally. We, too, discuss pharming as one of the potential abuses of dense, weak wifi deployments – exploitable in a different way but to a similar extent.

Mickens and Noble [43] propose a framework called *probabilistic queuing* to model the epidemic spreading in mobile environment, which aims to treat node mobility as top priority. Their simulations showed that the probabilistic queuing model could achieve more accurate prediction than standard Kephart-White framework in many cases. However, this work assumes random waypoint model for user movement and does not take into account realistic user mobility patterns.

Henderson *et al.* [33] analyzed extensive network traces from mature corporate WLANs and various university campuses and observed dramatic changes in wireless usage. Indeed, all these changes are favorable for the spread of a wifi worm. First, users now run a wide variety of applications such as peer-to-peer, multimedia and VoIP services, instead of the dominance of web traffic so there are higher chances of a worm exploitable vulnerability. Local traffic in the WLAN exceeds remote traffic, i.e. users within the same organization exchange data more than before. This would help the worm to detect and probe all wireless neighbors within its reach. The study also shows that wireless users are also surprisingly non mobile, half of which remain at home for 98% of time.

In a similar approach, Hsu and Helmy [34] found that there exists a preference of wireless user association: most users only visit a small portion of access points, i.e. the ratio of visited access points hardly changes even though popularity of WLANs increases by years – this is invariant user characteristics. There is a repetitive pattern of user association over days, i.e. there is a high probability that a user reappears at the same access point at a certain time every day. This is quantified as “network similarity index”. Therefore a mobile worm could distinguish itself from traditional internet worm by self-activating at the time where most mobile users are active. This is also contrary to the general assumption and over-simplification that users are always ON with no preference on association patterns; conventional randomly generated synthetic mobility models are insufficient. Another recent trend is that a mobile node stays online on average 87.68% of its life (i.e. its existence in the wireless network). That is to say, people now tend to use WLAN as a replacement for wired network and keep their laptops constantly connected (instead of old style of establishing only when needed). A modern paradigm shift from WLAN as temporary connection to always-on permanent connection. Macro mobility: users have small coverage in all environments (campus + corporate): typically only associate with 1.1% to 4.52% of total APs in their corporation. Each user has very few APs where it spends most of its online time.

Blinn *et al.* [21] monitored five weeks of Verizon wifi hotspot network in Manhattan. They observed that far

more cards associated to the network than logged into it. Most clients used the network infrequently and visited only few APs. Therefore hotspot are “locations visited occasionally” rather than “primary places of work”.

Kim *et al.* [37] extracted a mobility model from real user traces. Speed and pause time follow log-normal distribution and direction of movements closely related to road directions. Again, most of laptop clients are NOT very mobile, so this paper relied on VoIP users to extract mobility model. The type of mobile device being used can influence its user’s mobility: a laptop would tend to tie its user to his workplace whereas a PDA/VoIP user would move as he would normally. The reasons could be due to weight, size and nature of use of the device. A mobility model for laptop users should reflect relative weightage of immobility and mobility.

Staniford *et al.* [51] describe the risk to the Internet due to the ability of attackers to quickly gain control of vast numbers of hosts. They argue that controlling a million hosts can have catastrophic results because of the potential to launch distributed denial of service (DDoS) attacks and access any sensitive information that is present on those hosts. Their analysis shows how quickly attackers can compromise hosts using “dumb” worms and how “better” worms can spread even faster. In subsequent work [50], the same authors show how a worm using pre-compiled lists of IP addresses known to be vulnerable can infect one million hosts in half a second. They also envision a Cyber “Center for Disease Control” (CCDC) for identifying outbreaks, rapidly analyzing pathogens, fighting the infection, and proactively devising methods of detecting and resisting future attacks. The metropolitan wifi environment offers another opportunity for attacks to occur that may not be covered by defenses built for Internet worms. Our work also provides estimates of propagation speed similar to the above studies.

The issue of location privacy in a wireless setting has been examined in literature [35, 16, 31]. These system focus attention on protecting physical location privacy based against signal triangulation techniques and protecting against source location in sensor networks. More closely related to our work, is the work by Gruteser *et al.* [32]. The authors introduce the idea of short-lived disposable MAC addresses as a technique for the reduction of the effectiveness of location tracking. Our work shows that even in the presence of such techniques, user profiling can effectively track users in dense urban environments. Saponas *et al.* [47] describe a prototype surveillance system that can track people wearing the widely available Nike+iPod sensors. Tracknets could be exploited in similar scenarios to track people carrying any type of device whose traffic can be observed by wifi receivers, such as wifi-enabled smart-phones.

## 7 Concluding remarks

The increasing use of wireless technology and particularly wifi is likely to soon attract the attention of attackers, as attackers evolve and explore ways to exploit new technology to their advantage. This paper discusses a range of “modern” threats specifically tailored to metro-area wireless networks: wildfire worms that spread topologically due to infected hosts being able to carry the worm from one wireless LAN to another; large-scale wireless spoofing attacks that can be highly effective for phishing and spam campaigns; and malicious Tracknets that profile and track the whereabouts of wifi users. Such threats are greatly amplified by the increasingly dense deployment of wifi Access Points, and by the limited use of wireless security mechanisms such as 802.11i. Our results suggest that the density of large metropolitan areas has a profound impact on the severity of the threat.

Some specific contributions of this work include the modeling of fast, proximity-based worm propagation in metropolitan areas using real data from wardriving maps, wifi worm propagation using browser vulnerabilities, retrofitting of reactive mechanisms for wireless worm detection, spoofing defenses that are easy to implement, discussion of the whisper attack and defenses, and using RSS feeds to track users.

Our primary intention with this study is to raise awareness on the threats of wireless networks, specifically in densely populated areas, and to explore possible countermeasures. Much of the problem lies in the limited use of 802.11i. The wider deployment of 802.11i would reduce the risks significantly, but it would not completely eliminate them. More specifically, it would counter several instances of the spoofing threat; but it would only slow down, rather than mitigate wildfire worms; and it would not by itself eliminate the Tracknet threat, as MAC addresses remain unencrypted in 802.11i and other means of profiling may be possible.

Perhaps one of the main reasons behind the limited adoption of 802.11i is poor usability, as it involves configuration, and, once again, burdening users with yet another set of passwords or keys. Wider adoption requires convincing users that the extra trouble is worth it, by raising awareness on the risks of keeping wireless LANs open and unencrypted. We hope that our study contributes to this cause.

Improving usability of wireless security standards, if feasible, is another path to improving adoption, but until such adoption is achieved and to counter the remaining threats, we have also suggested a variety of countermeasures, which we have implemented and evaluated experimentally. Users may want to guard themselves against threats such as those described here, without having to take the cost of closing down their network using 802.11i or WEP.

## Acknowledgments

We thank S.P.T. Krishnan for assisting with the vulnerability exposure analysis, and K. Xinidis for his implementation of the basic OpenWRT-based defense infrastructure. We also thank Jonathan M. Smith, Pat Lincoln, Phil Porras, Angelos Keromytis, Michalis Polychronakis, Michael Nguyen and Lee Han Boon for extremely valuable discussions and feedback on this effort.

## References

- [1] Aircrack: a set of tools for auditing wireless networks. <http://freshmeat.net/projects/aircrack/>.
- [2] DShield.org, Distributed Intrusion Detection System. <http://www.dshield.org>.
- [3] Honeynet project. <http://www.honeynet.org>.
- [4] National vulnerability database, united states. <http://nvd.nist.gov>.
- [5] OpenWRT Project. <http://openwrt.org/>.
- [6] Securityfocus.com, vulnerabilities. <http://www.securityfocus.com/vulnerabilities>.
- [7] UMich wireless usage. <http://www.itcom.itd.umich.edu/wireless/stats/yr2006/02/campus.html>.
- [8] WEP: Dead Again. <http://www.securityfocus.com/infocus/1814#aircrack>.
- [9] Wifitap: proof of concept for communication over WiFi networks using traffic injection. [http://sid.rstack.org/index.php/Wifitap\\_EN](http://sid.rstack.org/index.php/Wifitap_EN).
- [10] Wireless Geographic Logging Engine. <http://www.wigle.net/>.
- [11] A year of bugs. <http://bcheck.scanit.be/bcheck/page.php?name=STATS2004&page=3>.
- [12] The Spread of the Sapphire/Slammer Worm. <http://www.caida.org/publications/papers/2003/sapphire/sapphire.html>, February 2003.
- [13] WMF exploitation. <http://www.f-secure.com/weblog/archives/archive-122005.html>, Dec. 2005.
- [14] E. Anderson, K. Eustice, S. Markstrum, M. Hanson, and P. Reiher. Mobile contagions: Simulation of infection and disease. In *Symposium on Measurement, Modeling, and Simulation of Malware*, June 2005.
- [15] W. A. Arbaugh, N. Shankar, and Y. J. Wan. Your 802.11 wireless network has no clothes. In *IEEE Wireless Communications*, 2001.
- [16] P. Bahl and V. N. Padmanabhan. RADAR: An in-building RF-based user location and tracking system. In *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 775–784, 2000.
- [17] M. Bailey, E. Cooke, F. Jahanian, J. Nazario, and D. Watson. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In *Proceedings of the 12<sup>th</sup> ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, pages 167–179, February 2005.
- [18] R. Beverly and S. Bauer. The spoofer project: Inferring the extent of source address filtering on the internet. In *Proceedings of USENIX Steps to Reducing Unwanted Traffic on the Internet (SRUTI) Workshop*, pages 53–59, July 2005.
- [19] R. A. Beyah, C. L. Corbett, and J. A. Copeland. The case for collaborative distributed wireless intrusion detection systems. In *IEEE International Conference on Granular Computing*, May 2006.

# On Web Browsing Privacy in Anonymized NetFlows

S. E. Coull\*      M. P. Collins<sup>†</sup>      C. V. Wright\*      F. Monroe\*      M. K. Reiter<sup>†</sup>

\*Johns Hopkins University  
{coulls,cwright,fabian}@cs.jhu.edu

<sup>†</sup>Carnegie Mellon University  
mcollins@cert.org, reiter@cmu.edu

## Abstract

Anonymization of network traces is widely viewed as a necessary condition for releasing such data for research purposes. For obvious privacy reasons, an important goal of trace anonymization is to suppress the recovery of web browsing activities. While several studies have examined the possibility of reconstructing web browsing activities from anonymized packet-level traces, we argue that these approaches fail to account for a number of challenges inherent in real-world network traffic, and more so, are unlikely to be successful on coarser NetFlow logs. By contrast, we develop new approaches that identify target web pages within anonymized NetFlow data, and address many real-world challenges, such as browser caching and session parsing. We evaluate the effectiveness of our techniques in identifying front pages from the 50 most popular web sites on the Internet (as ranked by alexa.com), in both a closed-world experiment similar to that of earlier work and in tests with real network flow logs. Our results show that certain types of web pages with unique and complex structure remain identifiable despite the use of state-of-the-art anonymization techniques. The concerns raised herein pose a threat to web browsing privacy insofar as the attacker can approximate the web browsing conditions represented in the flow logs.

## 1 Introduction

Recently, significant emphasis has been placed on the creation of anonymization systems to maintain the privacy of network data while simultaneously allowing the data to be published to the research community at large [23, 24, 17, 9, 22]. In general, the goals of anonymization are (i) to hide structural information about the network on which the trace is collected, so that disclosing the anonymized trace does not reveal private information about the security posture of that network,

and (ii) to prevent the assembly of behavioral profiles for users on that network, such as the web sites they browse.

Our goal in this paper is to evaluate the strength of current anonymization methodology in achieving goal (ii). Specifically, we focus on providing a realistic assessment of the feasibility of identifying individual web pages within anonymized NetFlow logs [4]. Our work distinguishes itself from prior work by operating on flow-level data rather than packet traces, and by carefully examining many of the practical concerns associated with implementing such identification within real network data. Previous work has focused on methods for web page identification within encrypted or anonymized packet trace data utilizing various packet-level features, such as size information, which cannot be readily scaled to flow-level data. Rather than assume the presence of packet-level information, our work instead focuses on the use of flow-level data from NetFlow logs to perform similar identification. Since NetFlow data contains a small subset of the features provided in packet traces, we are able to provide a general method for identifying web pages within both packet trace and NetFlow data. Also, use of NetFlow data is becoming more commonplace in network and security research [13, 21, 33, 5].

More importantly, our primary contribution is a rigorous experimental evaluation of the threat that web page identification poses to anonymized data. Though previous work has provided evidence that such identification is a threat, these evaluations do not take into account several significant issues (e.g., dynamic web pages, browser caching, web session parsing, HTTP pipelining) involved with the application of deanonymizing techniques in practice. To overcome these obstacles to practical identification of web pages, we apply machine learning techniques to accommodate variations in web page download behavior<sup>1</sup>. Furthermore, our techniques can parse and identify web pages even within multiple interleaved flows, such as those created by tabbed browsing, with no additional information. The crux of our identi-

cation method lies in modeling the web servers which participate in the download of a web page, and using those models to find the corresponding servers within anonymized NetFlow data. Since the behavior of each server, in terms of the flows they serve, is so dynamic, we apply kernel density estimation techniques to build models that allow for appropriate variations in behavior.

Simply finding web servers is not enough to accurately identify web pages, however. Information such as the order in which the servers are contacted, and which servers are present can have significant impact on the identification of web pages. In fact, the ordering and presence of these servers may change based on various download scenarios, such as changes in browser cache or dynamic web page content. To capture these behaviors, we formalize the game of “20 Questions” as a binary Bayes belief network, wherein questions are asked to narrow the possible download scenarios that could explain the presence of a web page within the anonymized data. As such, our approach to web page identification begins with identifying likely servers and then employs the binary Bayes belief network to determine if those servers appropriately explain the presence of the targeted web page within the data.

Lastly, the evaluation of our techniques attempts to juxtapose the assumptions of closed world scenarios used in previous work to the realities of identifying web pages in live network data. The closed world evaluation of data collected through automated browsing scripts within a controlled environment was found to perform well — detecting approximately 50% of the targeted web pages with less than 0.2% false detections. In more realistic scenarios, however, true detection and false detection rates varied substantially based upon the type of web page being identified. Our evaluation of data taken through controlled experiments and live network captures shows that certain types of web pages are easily identifiable in real network data, while others maintain anonymity due to false detections or poor true detection rates. Additionally, we show the effects of locality (i.e., different networks for collecting training and testing data) on the detection of web pages by examining three distinct datasets taken from disparate network environments. In general, our results show that information leakage from anonymized flow logs poses a threat to web browsing privacy insofar as an attacker is able to approximate the basic browser settings and network conditions under which the pages were originally downloaded.

## 2 Background and Related Work

Network trace anonymization is an active area of research in the security community, as evidenced by the ongoing development of anonymization methods

(e.g., [9, 23, 30]) and releases of network data that they enable (e.g., [26, 7]). Recently, several attacks have been developed that illustrate weaknesses in the privacy afforded by these anonymization techniques. In particular, both passive [6] and active attacks [2, 3] have shown that deanonymization of public servers and recovery of network topology information is possible in some cases. Until now, however, an in-depth examination of the extent to which the privacy of web browsing activities may also be at risk has been absent.

It would appear that existing approaches for inferring web browsing activities within encrypted tunnels [19, 32, 11, 1, 18, 8]) would be directly applicable to the case of anonymized network data—in both cases, payload and identifying information (e.g., IP addresses) for web sites are obfuscated or otherwise removed. These prior works, however, assume some method for unambiguously identifying the connections that constitute a web page retrieval. Unfortunately, as we show later, this assumption substantially underestimates the difficulty of the problem as it is often nontrivial to unambiguously delineate the flows that constitute a single page retrieval. The use of NetFlow data exacerbates this problem. Furthermore, as we show later, there are several challenges associated with the modern web environment that exacerbates the problem of web page identification under realistic scenarios.

To our knowledge, Koukis et al. [14] present the only study of web browsing behavior inference within anonymized packet traces, which anticipates some of the challenges outlined herein. In their work, however, the authors address the challenges of parsing web page downloads from packet traces by using packet inter-arrival times to delineate complete sessions. Though this delineation can be successful in certain instances, there are several cases where time-based delineation alone will not work (e.g. for interleaved browsing). In this paper, we address several challenges beyond those considered by Koukis et al. and provide a more in-depth evaluation that goes further than their exploratory work. Moreover, our work differs from all prior work on this problem (of which we are aware) in that it applies to flow traces, which offer far coarser information than packet traces.

## 3 Identifying Web Pages in Anonymized NetFlow Logs

The anonymized NetFlow data we consider consists of a time-ordered sequence of records, where each record summarizes the packets sent from the server to the client within a TCP connection. These unidirectional flow records contain the source (server) and destination (client) IP addresses, the source and destination port

numbers, timestamps that describe the start and end of each TCP connection, and the total size of the traffic sent from the source to the destination in the flow (in bytes). The NetFlow format also contains a number of other fields that are not utilized in this work. For our purposes, we assume that the anonymization of the NetFlow log creates consistent pseudonyms, such as those created by prefix-preserving anonymization schemes [9, 23], for both the source and destination IP addresses in these records. Furthermore, we assume that the NetFlow data faithfully records TCP traffic in its entirety.

The use of consistent pseudonym addresses allows us to separate the connections initiated from different hosts, thereby facilitating per host examination. Additionally, we assume that port numbers and sizing information are not obfuscated or otherwise altered to take on inconsistent values since such information is of substantial value for networking research (e.g., [10, 29, 12]). The unaltered port numbers within the flows allow us to filter the flow records such that only those flows originating from port 80 are examined<sup>2</sup>.

Initially, we also assume that *web browsing sessions* (i.e., all flows that make up the complete download of a web page) can be adequately parsed from the NetFlow log. A similar assumption is made by Sun et al. [32] and Liberatore et al. [18]. Though previous work has assumed that web browsing session parsing algorithms are available, accurate web session parsing is, in fact, difficult even with packet traces and access to payload information [31, 15]. In §6, we return to the difficulty of parsing these sessions from real anonymized network data. By adopting the assumption (for now) that accurate web browsing session parsing can be done, it becomes possible to parse the complete NetFlow data into non-overlapping subsequences of flow records, where each subsequence represents a single, complete web browsing session for a client. Given the subsequent client web browsing sessions, our goal is to extract features that uniquely identify the presence of target web pages within the anonymized NetFlow data, and model their behavior in a manner that captures realistic browsing constraints.

### 3.1 Feature Selection

The most intuitive feature for discovering web pages in the anonymized NetFlow data is the sequence of flow sizes observed during a complete web browsing session. Each flow in the web browsing session is represented by an *index number* indicating its ordering in the session, and an associated flow size indicating the amount of data transferred during the flow. Naïvely, one would expect that the use of flow size, index pairs would suffice as a good distinguisher for web page identification. However, as Figure 1(a) shows, this is not the case. For instance,

notice that the front page of *msn.com* is fairly inconsistent in the number and size of flows, and there is a significant amount of overlap even among only these three examples. Since we are examining flows, the number of flows and their associated sizes are dependent on the manner in which the client requests objects, such as pictures or text. In many cases, the sequence in which the objects are downloaded may change due to dynamic web content, or the state of the client's browser cache may cause certain objects to be excluded. These changes to the client's download behavior cause *object drift* within the flows, where web page objects are downloaded in different flows or not downloaded at all. As a result, the number of flows and their respective sizes can vary widely, and are therefore a poor indicator of the identity of the web page in question.

An important observation regarding this inconsistency is that the size of any flow is regulated by the cumulative size of all the objects downloaded for the web page, less the size of all objects downloaded in prior flows. If a large flow early in the browsing session retrieves a significant number of objects, then the subsequent flows must necessarily become smaller, or there must be fewer flows overall. Conversely, a session of many small flows must necessarily require more flows overall. In fact, if we examine the cumulative perspective of web page downloads in Figure 1(b), we find that not only are these sites distinguishable, but that they take consistent paths toward their target cumulative size.

The existence of such paths and the inherent connection between flow size, index number, and cumulative size motivates the use of all three features in identifying web pages. These three features can be plotted in 3-dimensional space, as shown in Figure 2(a), and the path taken in this 3-dimensional space indicates the behavior exhibited by the download of objects for a complete web browsing session. Figure 2(b) shows an example of web browsing session paths for the front pages of both *yahoo.com* and *msn.com* overlaid on the set of points taken over many web browsing sessions of *msn.com*'s front page. Clearly, the path taken by *yahoo.com* is distinct from the set of points generated from web browsing sessions of *msn.com*, while the *msn.com* path remains similar to past web browsing sessions.

**Server sessions** The use of flow size, index number, and cumulative size information can be further refined by considering the sequence of flows created from each web server in the web browsing session, which we denote as a *server session*. Notice that when we separate the flows for *msn.com* by the server that produced them, each server occupies a very distinct area of the 3-dimensional space, as shown in Figure 3. This refinement offers two benefits in identifying web pages.

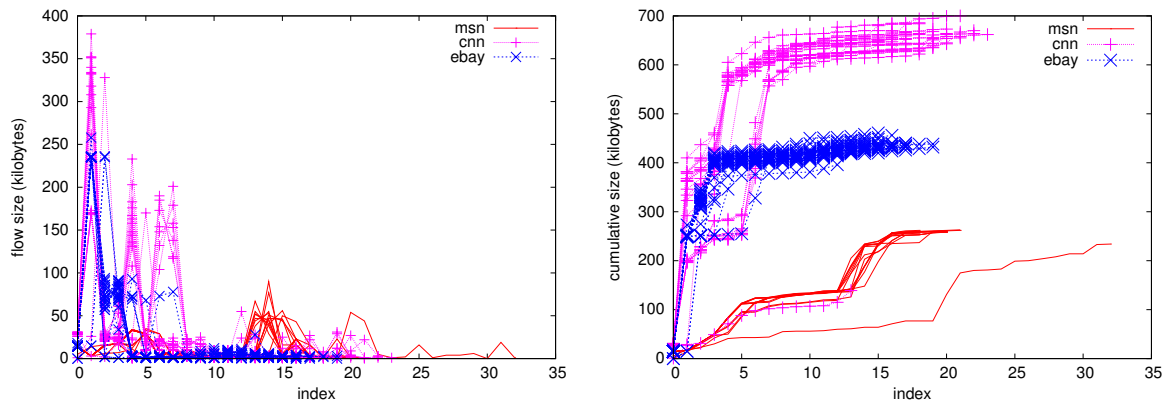


Figure 1: (a) Sequential and (b) cumulative views of page loads for *msn.com*, *cnn.com*, and *ebay.com* from a single client

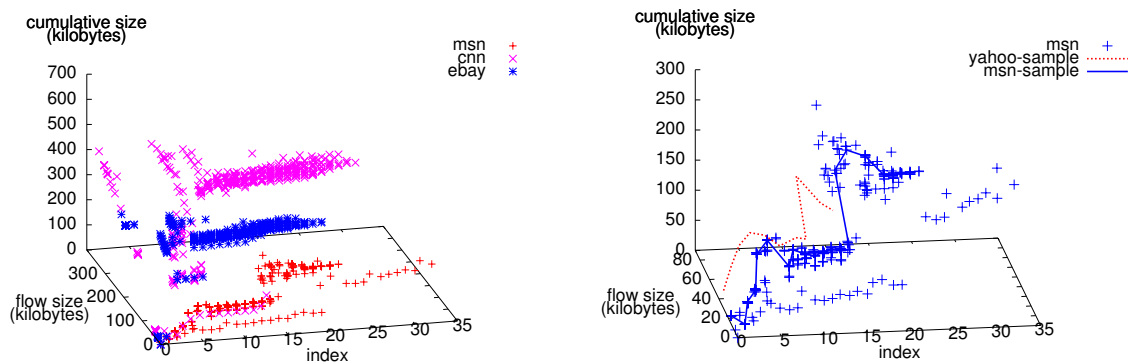


Figure 2: (a) 3-D view of page loads for *msn.com*, *cnn.com*, and *ebay.com* from a single client; (b) Regions for *msn.com* compared to sequences of *yahoo.com* and *msn.com* as downloaded by a single client

First, by abstracting the web browsing sessions to consist of individual server sessions, we can use the presence or absence of servers and their relative ordering to further differentiate web pages. The ordering of these web servers provides useful information about the structure of the web page since there is often a dependency between objects within the web page. For instance, the HTML of a web page must be downloaded before any other objects, and thus the first server contacted must be the primary web server. Second, by refining our flow information on a per server basis, we can create a fine grained model of the behavior of the web browsing session. If done correctly, the problem of identifying a web page within anonymized NetFlow data can be reduced to one of identifying the servers present within a given web browsing session based on the path created by the flows they serve, and the order in which the servers are contacted.

**Logical servers** Intuitively, we could simply use the flows served by each distinct web server IP address (which we refer to as a *physical* server) to create the 3-dimensional space that describes the expected behavior of that physical server in the web browsing session. However, the widespread use of Content Delivery Networks (CDNs) means that there may be hundreds of distinct physical web servers that serve the same web objects and play interchangeable roles in the web browsing session. These farms of physical servers can actually be considered to be a single *logical* server in terms of their behavior in the web browsing session. Therefore, the 3-dimensional models we build are derived from the samples observed from all physical servers in the logical server group.

Of course, the creation of robust models for the detection of web pages requires that the data used to create the models reflect realistic behaviors of the logical servers and the order in which they are contacted. There are a number of considerations which may affect the ability

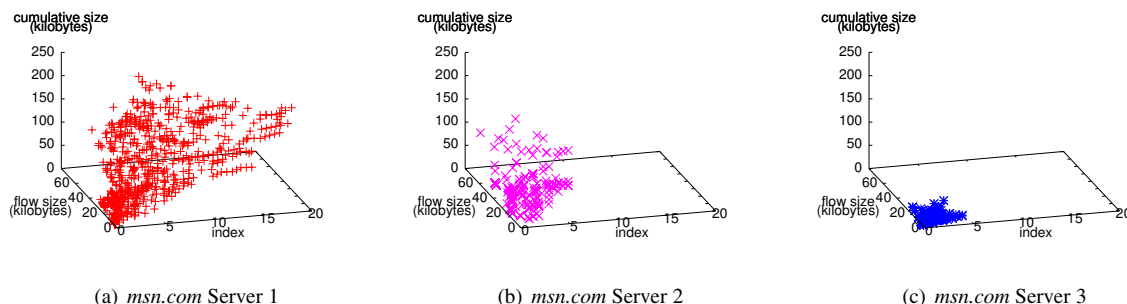


Figure 3: 3-D view of *msn.com* separated by server as observed by a single client

of data to accurately predict the behavior of web page downloads. These considerations are especially important when an attacker is unable to gain access to the same network where the data was collected, or when that data is several months old. Liberatore et al. have shown that the behavioral profiles of web pages, even highly dynamic web pages, remain relatively consistent even after several months [18], though the effects of web browser caching behavior and the location where the network data was captured have not yet been well understood.

## 4 An Automated Classifier for Web Pages in NetFlows

In this section, we address the problem of building automated classifiers for detecting the presence of target web pages within anonymized NetFlow data. Through the use of features discussed in §3, we create a classifier for each web page we wish to identify. The classifier for a target web page consists of the 3-dimensional spaces for each of its logical servers, which we formalize by using (i) kernel density estimates [28], and (ii) a series of constraints for those logical servers, formalized by a binary Bayes belief network [20]. The goal of the classifier is to attempt to create a mapping between the physical servers found in the anonymized web browsing session and the logical servers for the target web page, and then to use the mapping to evaluate constraints on logical servers for the web page in question. These constraints can include questions about the existence of logical servers within the web browsing session, and the order in which they are contacted by the client. If the mapping meets the constraints for the given web page, then we assume that the web page is present within the web browsing session; otherwise, we conclude it is not.

There are several steps, illustrated in Figure 4, that must be performed on the anonymized NetFlow logs in order to accurately identify web pages within them. Our first step is to take the original NetFlow log and parse

the flow records it contains into a set of web browsing sessions for each client in the log. Recall that our initial discussion assumes the existence of an efficient and accurate algorithm for parsing these web browsing sessions from anonymized NetFlow logs. These web browsing sessions, by definition, consist of one or more physical server sessions, which are trivially parsed by partitioning the flow records for each client, server pair into separate physical server sessions. The physical server sessions represent the path taken within the 3-dimensional space (i.e., flow size, cumulative size, and index triples) when downloading objects from the given physical server. At this point, we take the paths defined by each of the physical servers in our web browsing session, and see which of the logical servers in our classifier it is most similar to by using kernel density estimates [28]. Therefore, a given physical server is mapped to one or more logical servers based on its observed behavior. This mapping indicates which logical servers may be present within our web browsing session, and we can characterize the identity of a web page by examining the order in which the logical servers were contacted using a binary belief network. If we can satisfy the constraints for our classifier based upon the logical servers present within the web browsing session, then we hypothesize that an instance of the web page has been found. In §4.1 and §4.2, we discuss how the kernel density estimates and binary belief networks are created, respectively.

### 4.1 Kernel Density Estimation

In general, the kernel density estimate (KDE) [28] uses a vector of samples,  $S = \langle s_1, s_2, \dots, s_n \rangle$  to derive an estimate for a density function describing the placement of points in some  $d$ -dimensional space. To construct a KDE for a set of samples, we place individual probability distributions, or *kernels*, centered at each sample point  $s_i$ . In the case of Gaussian kernels, for instance, there would be  $n$  Gaussian distributions with means of  $s_1, s_2, \dots, s_n$ , respectively. To control the area covered by each distribution, we can vary the so-called *bandwidth* of the ker-

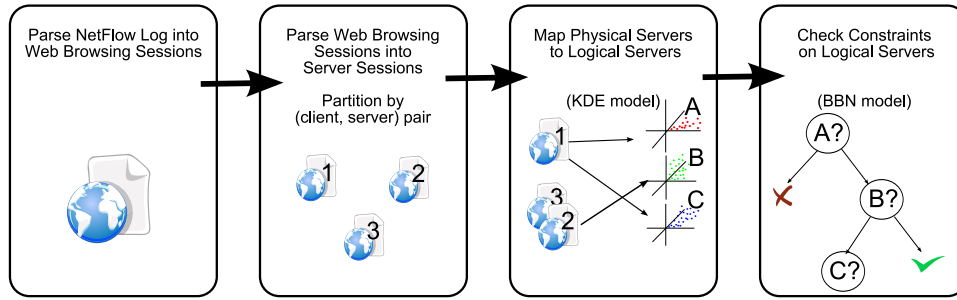


Figure 4: General overview of our identification process

nel. For a Gaussian kernel, its bandwidth is given by the variance (or covariance matrix) of the distribution. Intuitively, a higher bandwidth spreads the probability mass out more evenly over a larger space.

Unfortunately, determining the appropriate bandwidth for a given set of data points is an open problem. One approach that we have found to produce acceptable results is to use a “rule of thumb” developed by Silverman [28] and refined by Scott [27]. The bandwidth is calculated as  $H_{\Delta} = N^{-1/(d+4)}\sigma_{\Delta}$ , for each dimension  $\Delta = 1 \dots d$ , where  $N$  is the number of kernels,  $d$  is the number of dimensions for each point, and  $\sigma_{\Delta}$  is the sample standard deviation of the  $\Delta$  dimension from the sample points in  $S$ . The primary failing of this heuristic is its inability to provide flexibility for multi-modal or irregular distributions. However, since this heuristic method provides adequate results for the problem at hand, we forego more complex solutions at this time.

Once the distributions and their associated bandwidths have been placed in the 3-dimensional space, we can calculate the probability of a given point,  $t_j$ , under the KDE model as:

$$P(t_j) = \frac{1}{n} \sum_{i=0}^{n-1} P_{s_i}(t_j) \quad (1)$$

where  $P_{s_i}(t_j)$  is the probability of point  $t_j$  under the kernel created from sample point  $s_i$ .

#### 4.1.1 Application to Web Page Identification

To apply our anonymized NetFlow data to a KDE model, we take the set of paths defined by the triples of flow size, cumulative size, and index in each of the physical server sessions of our training data and use them as the sample points for our kernels. We choose the Gaussian distribution for our kernels because it allows us to easily evaluate probabilities over multiple dimensions. The bandwidth of these distributions is calculated as described previously, except that for the flow size and cumulative size dimensions, we take the average standard deviation across all index values as the bandwidth. Furthermore, we bound the bandwidth in each dimension such that it

is always  $\geq 1$  to allow for some minimum amount of variability.

To evaluate an anonymized physical server session on a particular KDE model, we simply evaluate each point in the path for that physical server session using Eqn. 1, and calculate the total probability of the given physical server session,  $t$ , as:

$$P(t) = \prod_{j=0}^{m-1} P(t_j) \quad (2)$$

where  $t_j$  is the  $j^{th}$  point in the physical server session  $t$ , and  $m$  is the number of flows in  $t$ . For classification, we consider any physical server session whose path has a non-zero probability (from Eqn. 2) under the given model to be a mapping between the logical server represented by the model and the physical server session being evaluated. Of course, it may be possible for physical server session  $t$  to follow a path that matches portions of several disjoint paths in the KDE model without exactly matching any paths in their entirety. Consequently, the path would achieve a non-zero probability despite the fact that it is not similar to any of the paths in the model. To prevent such situations from occurring, we apply linear interpolation to each pair of points representing consecutive flow indices on a path to create sample points at half index intervals.

The use of path probabilities alone, however, is insufficient in uniquely describing the behavior of the logical server. To see why, consider the case where we have a model for a logical server which typically contains ten or more total flows. It may be possible for a much smaller physical server session with one or two flows to achieve a non-zero probability despite the fact that there clearly was not an adequate amount of data transferred. To address this, we also create a KDE model for the final points in each sample path during training, denoted as *end points*. These end points indicate the requisite cumulative size and number of flows for a complete session with the given logical server. As before, we create distributions around each sample end point, and calculate the

probability of the physical server session's end point by applying Eqn. 1. Any anonymized physical server session which has a non-zero probability on both their path and their end points for a given logical server model is mapped to that logical server.

### Automatically Building Logical Server KDE Models

To create our logical server models, we use two heuristics to group physical servers into logical server groups. First, if two physical servers in our training data use the same hostname and serve the exact same HTTP URL, we can assume they are the same logical server and their sample points can be merged into a single KDE model. Since we are in control of our training data, we can collect packet traces to find URL and hostname information before converting the data to NetFlow format to create the paths that will make up our KDE models. It is often the case, however, that different hostnames are used among physical servers in the same logical server group, and this may prevent some of the physical servers in our training data from being placed into the correct logical server groups.

To address this, we apply a second heuristic that merges these remaining physical servers by examining the behavior exhibited by their KDE models. If a randomly selected path and its end point from a given physical server's training data achieves non-zero probability on the KDE model of another physical server, then those two physical servers can be merged into a single logical server. The combination of these two heuristics allow us to reliably create KDE models that represent the logical servers found in the web browsing session. By applying the points found in an anonymized physical server session to each of the KDE models for a given web page, we can create candidate mappings from the anonymized physical server to the logical servers for the target web page.

## 4.2 Binary Bayes Belief Networks

As discussed earlier, we formalize the constraints on the logical servers using a binary Bayes belief network (BBN). In a typical Bayes network, nodes represent events and are inter-connected by directed edges which depict causal relations. The likelihood that a given event occurs is given by the node's probability, and is based on the conditional probability of its ancestors. In the binary Bayes belief network variant we apply here, we simply use a binary belief network where events have boolean values and the causal edges are derived from these values [20].

An example of a binary belief network is given in Figure 5, where the probability of event  $y$  is conditioned upon event  $\neg x$ . One way of thinking of this network is

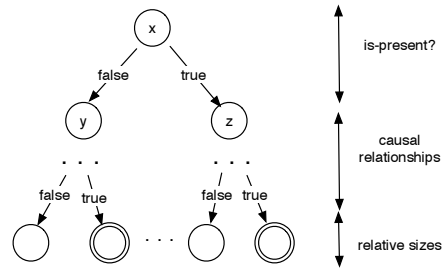


Figure 5: Example BBN

as a strategy for the game of “20 Questions” where the player attempts to identify an object or person by asking questions that can only be answered with ‘Yes’ or ‘No’ responses. Our binary belief network is simply a formalization of this concept (though we are not limited to ask 20 questions), where the answer to any question dictates the best strategy for asking future questions.

To create the belief network, we first decide upon a set of questions (or events) that we would like to evaluate within the data. In the context of web privacy, these events relate to the existence of logical servers, their causal relationships, and cumulative size. The belief network can be created *automatically* by first examining all possible existence and ordering events that occur within the training data. Next, from this set of events, we can simply select the event whose probability of being True in the training data is highest among all events. Having done so, the training data can then be partitioned into two groups: one group whose data has the value True for the selected event, and another whose value is False for that event. The selected event is then removed from the set of possible events and each partition of training data now selects another event from the remaining set whose probability on their respective data is highest.

This partitioning process is repeated recursively, allowing each branch to grow independently. A given branch halts its recursion when its conditional probability for an event is  $< \epsilon$ . The conditional probability threshold,  $\epsilon$ , indicates the percentage of the training data that remains at a given leaf node, and therefore we stop our recursion before the tree becomes overfitted to our training data. Any leaf node that halts recursion with some amount of training data remaining is considered as an accepting node, and all other leaf nodes are labeled as rejecting nodes. Accepting nodes implement one additional check to ensure that the total size of all flows in the web browsing session is within  $\pm 10\%$  of the total sizes observed during training.

## 5 A Closed-world Evaluation

To gauge the threat posed by the our web page identification techniques—and to place our results in context with prior work—we first provide an evaluation under a clean, closed world testing model. Prior work on this topic also focuses on the evaluation of identification techniques based on a controlled network environment, browsing a set of target web pages across an encrypted tunnel [32], through a proxy server [18], or within anonymized packet traces [14]. Each of these works, with the exception of [14], also assume that the web browsing session can be easily parsed from the stream of packets crossing the encrypted tunnel or proxy server. In what follows, we also adopt this assumption for this particular evaluation, though we will re-visit the inherent challenges with web browsing delineation in §6.

In short, our initial evaluation is considered under controlled environments similar to past work, but with two notable differences. First, in the scenarios we examine, there is substantially less data available to us than at the packet trace level; recall that NetFlows aggregate all packets in a flow into a single record. Second, rather than assuming that the client’s browser cache is turned off, we attempt to simulate the use of caching in browsers in our training and testing data. The simulation of browser caching behavior was implemented by enabling the default caching and cookie policies within Mozilla Firefox™, and browsing to the sites in our target set at random. Of course, this method of cache simulation is not entirely realistic, as the probability of a cache hit is directly proportional to the frequency with which the user browses that web site. However, in lieu of making any assumptions on the distribution of web browsing for a given user, we argue that for the comparison at hand, the uniform random web browsing behavior provides an adequate approximation.

**Data Collection** The data for our closed world evaluation was collected with the use of an automated script that used Firefox to randomly visit selected web pages from a set of target pages (with Adobe Flash and Javascript enabled). The web browser was set to the default caching and cookie policies to ensure the most realistic behavior possible in such a closed world environment. Specifically, the script first initiated a new Firefox instance, and opened new tabs within the single Firefox instance for each new web page visited. While these web pages were not loaded in parallel, several web sites automatically refresh themselves at given intervals, thus adding noise to our data whenever they appeared among one of the tabs of the active Firefox instance. Once four web pages were opened in the current Firefox instance, the browser was closed gracefully to allow the cache to

flush to disk, and a new Firefox instance was loaded to continue the random browsing. For each visit to a web page, we captured the packets for that web browsing session and recorded it to a separate trace. The packet captures were then converted into NetFlow logs by creating single flow records for each TCP connection in the session. Notice that the use of an automated browsing script allowed us to cleanly delineate between browsing sessions, as well as to simulate cache behavior through random browsing.

Our target pages were the front pages of the top 50 most popular sites as ranked by *alexa.com*. Additionally, we also collected information about the front pages of sites ranked 51-100 on the *alexa.com* list for use in providing robust evaluation of the false detection rates of our technique. Though we have chosen to evaluate our techniques on the top 100 sites, there is nothing inherent in their structure that differentiates them from other web pages. In fact, the same techniques are equally applicable in targeting any web page of the attacker’s choosing.

The web pages were retrieved by running the automated browsing script on a host within the Johns Hopkins University network for a total of four weeks, creating a total of 18,525 web browsing sessions across all 100 web pages in our list of web pages. From this data, we select the first 90 web browsing sessions of our target web pages (i.e., those within the top 50 of the *alexa.com* ranking) as the training data for the creation of the kernel density estimate (see §4.1) and binary Bayes belief network models (see §4.2) that make up the profiles for each target web page. The remaining sessions are used as test data and are anonymized by replacing IP addresses within the NetFlow data with prefix-preserving pseudonyms according to the techniques described by [23]. Notice that since we assume that the web browsing sessions are easily parsed, we can simply use each web browsing session in our test data directly to determine if that web browsing session can be identified as any of the 50 web pages in our target set using the techniques described in §3.

**Results** The results for this evaluation are given in Table 1. The analysis shows that our web page identification method performs reasonably well in the closed world environment. Though the overall true detection rate is only 48%, its associated false detection rate is exceptionally low at only 0.18% across all web pages. For comparison, using random guessing to identify web pages would yield an overall true detection rate of only 2%. Moreover, keep in mind that under the goals of network data anonymization, *no* inference of browsing behavior should be possible.

For ease of exposition, we also partition the 50 target web pages into canonical categories based on the primary

Category	Examples	TD (%)	FD (%)
Other	<i>passport.net, statcounter.com</i>	95.33	0.16
Social Networking and Dating	<i>match.com, myspace.com</i>	64.59	0.11
Search Engines and Web Portals	<i>msn.com, google.com</i>	60.42	0.16
Reference	<i>imdb.com, wikipedia.org</i>	54.17	0.02
Media	<i>flickr.com, youtube.com</i>	52.82	0.42
Corporate	<i>microsoft.com, apple.com</i>	48.95	0.15
Shopping	<i>amazon.com, ebay.com</i>	39.22	0.00
News	<i>cnn.com, nytimes.com</i>	28.74	0.06
Job Search	<i>monster.com, careerbuilder.com</i>	26.73	0.00
Sports	<i>foxsports.net, mlb.com</i>	20.74	0.00
Overall		48.89	0.18

Table 1: True and false detection rates for canonical categories in closed world test

function of the web site. Notice that the performance of the canonical classes varies based on the dynamism of the contents in the web page. For instance, some of the more difficult categories in terms of true detection are those whose front page content changes frequently, e.g., *cnn.com*. Conversely, pages with simple, static content, like *passport.net* or *google.com*, can be identified reasonably well. Moreover, those web sites with simple layouts and little supporting infrastructure also tend to fare worst with respect to false detections, while complex, dynamic sites have few, if any, false detections. These initial results hint at the fact that the ability to reliably identify web pages is connected with the complexity and dynamism of the web page. In what follows, we examine whether these results hold under a more realistic examination based on real world browsing.

## 6 Considerations for the Real World

The closed world evaluation in the previous section made several assumptions about the attacker’s ability to parse web sessions and simulate caching behavior. Moreover, since both the training and testing data were collected at the same location, the effects of locality on the effectiveness of the identification techniques were not accounted for. These assumptions lead to a disconnect between the results of our closed world testing and those that can be expected in realistic attacks on anonymized data. In order to perform a rigorous evaluation of the *real* threats posed by such identification techniques, we must address several issues, including web browsing session parsing and caching behavior.

**Web Browsing Session Parsing** One of the biggest concerns with the closed world evaluation in §5 is that there is an implicit assumption that parsing web sessions from live network data is a simple and accurate task. There has been extensive work in attempting to parse

packet traces into web browsing sessions, yet much of this work requires access to plaintext payloads, and results show that this parsing is not completely accurate [16, 31, 15]. To our knowledge, there is no prior art on performing similar parsing on NetFlow data. Koukis et al. attempted to use a heuristic of packet inter-arrival times to delineate sessions in packet traces, but their techniques were only able to correctly identify 8% of the web browsing sessions—underscoring the difficulty of the problem [14].

Fortunately, our kernel density estimate (KDE) and binary Bayes belief network (BBN) models can be modified to overcome the challenges of web browsing session parsing without significant changes to our identification process. In our previous evaluation, we assumed that the KDE and BBN models were given a subsequence of the original NetFlow log that corresponded to a complete web browsing session for a single client. For our real world evaluation, however, we remove this assumption. Instead, to parse the NetFlow log, we assume that all flows of a given web browsing session are clustered in time, and partition the NetFlow log into subsequences such that the inter-arrival times of the flows in the partition is  $\leq \delta = 10$  seconds. This assumption is similar to that of Koukis et al. [14] and provides a coarse approximation to the web browsing sessions, but the resultant partitions may contain multiple web browsing sessions, or interleaved sessions.

Notice that we can simply use each of the physical servers within these partitions as input to the KDE models for a target web page to determine which logical servers may be present in the partition. Thus, we apply the flows from every physical server in our partitioned NetFlow data to the KDE models for our target page to create the logical server mappings. If a physical server in our partition does not map to a logical server, we ignore that physical server’s flows and remove it from the partition. Thus, by removing these unmapped physical

servers, we identify a candidate web browsing session for our target site. Since the BBN operates directly on the mappings created by the KDE models, we traverse the BBN and determine if the web page is present based on the physical servers that were properly mapped. This technique for finding web browsing sessions is particularly robust since it can find multiple web pages within a single partition, even if these web pages have been interleaved by tabbed browsing.

**Browser Cache Behavior** Another serious concern in our closed world evaluation is the variability of web browsing session behavior due to the client's browser cache. In our closed world evaluation, we created our models from data collected by an automated script that randomly browsed the front pages from among the top 100 sites according to *alexa.com*. The use of uniform random browsing with the default cache policy, however, does not accurately reflect the objects that would be cached by real clients. In reality, the client's browser cache would tend to hold more objects from the most frequently visited web pages, making the cache states highly specific to the client. Clearly, using our simulated caching data alone is not enough to create models that are able to detect both frequently and infrequently visited sites. To alleviate this shortcoming, we create a second set of training data by setting the browser's cache limit to 1.5GB. With such a large browser cache, objects should not be evicted from the cache even when we perform our random browsing, thereby allowing us to gain information about web browsing behaviors for our target sites when they are viewed frequently. The training data that we use to create our models now consists of 90 web browsing sessions of simulated cache data, and 64 browsing sessions of unlimited cache data for each target site. The procedure for building our models remains the same, except we now use the flow records from both cache scenarios.

**Results** To provide a more realistic evaluation of the threat our identification techniques pose to anonymized NetFlow data, we re-examine its performance on three distinct datasets. First, we use the testing data from our closed world evaluation to measure the effect that the introduction of unlimited cache data and web session parsing have on the performance of our technique. Second, we capture web browsing sessions from different network providers in Maryland, and in Pennsylvania. By comparing the performance of our technique on these three datasets, we can glean insight into the effects of locality on the success of attacks on anonymized NetFlow data.

The effects that the changes to our models have on the performance of our technique are shown in Table

2. Clearly, the false detection rate increases substantially, but the true detection rate also increases. As in the closed world scenario, we find that the web pages with constantly changing content are more difficult to detect than static web pages, and that those sites with complex structure (i.e., many logical servers, and many flows) achieve a significantly lower false detection rate than those sites with simple structure. The substantial change in performance can be explained by the relaxation of the BBN constraints to allow for web browser session parsing. This relaxation allows any web browsing session where a subset of physical servers meets the remaining constraints to be identified, thereby causing the increase in both true detection and false detection rate. A more detailed analysis of the implications of these effects is provided in §7.

It is often the case that published network data is taken at locations where an attacker would not have access to the network to collect training data for her models, and so we investigate the effect that the change in locality has on the performance of our technique. The results in Table 2 show that there is, indeed, a drop in performance due to changes in locality, though trends in true detection and false detection rates still hold. In our evaluation, we noticed that the Johns Hopkins data used to train our web page models included a web caching server that caused significant changes in the download behavior of certain web pages. These changes in behavior in turn explain the significant difference in performance among data collected at different localities. It would appear that these results are somewhat disconcerting for a would-be attacker, since she would have to generate training data at a network that was different from where the anonymized data was captured. However, she could make her training more robust by generating data on a number of networks, perhaps utilizing infrastructure such as PlanetLab [25], though the effects of doing so on the performance of the technique are unknown. By including web page download behavior from a number of networks, she can ensure that the KDE and BBN models for each target web page are robust enough to handle a variety of network infrastructures.

## 7 A Realistic Threat Assessment

Finally, we provide a threat assessment by applying our technique to live data collected from a public wireless network at Johns Hopkins University Security Institute over the course of 7 days. From this data, we examine the expected real world accuracy of our techniques and discuss the features that make some web pages prime targets for identification.

Category	TD (%)	FD (%)	Maryland		Pennsylvania	
			TD ( $\Delta$ )	FD ( $\Delta$ )	TD ( $\Delta$ )	FD ( $\Delta$ )
Other	100.00	10.71	-40.00	+1.98	-6.25	+10.74
Search/Web Portals	94.29	13.92	-35.47	+3.92	-26.60	-0.45
Social Network/Dating	75.75	9.02	-15.75	+4.93	+13.61	+9.95
Media	75.71	30.61	-3.71	+2.00	-30.97	+5.75
Corporate	75.00	11.64	-37.50	+2.55	-43.09	+3.06
Job Search	72.50	1.81	-47.50	+1.10	-44.91	+7.94
Shopping	71.00	4.89	-17.67	+3.55	-40.05	+1.37
News	70.71	2.50	-41.54	+0.15	-35.00	-0.14
Reference	67.00	14.64	-25.82	+9.59	+1.52	+19.13
Sports	66.67	26.57	-9.53	-6.23	+4.16	-11.87
Overall	75.58	13.28	-26.86	+1.47	-24.39	+4.59

Table 2: True and false detection rates for canonical categories in JHU data, and comparison to remote datasets

Category	Web Page	TD (%)	FD (%)
Reference	<i>imdb.com</i>	100.00	13.1
News	<i>nytimes.com</i>	0.00	0.06
	<i>digg.com</i>	61.76	0.35
	<i>washingtonpost.com</i>	9.09	0.01
	<i>cnn.com</i>	44.00	8.30
	<i>weather.com</i>	0.00	2.27
Search/Web Portals	<i>google.com</i>	28.57	22.31
	<i>msn.com</i>	0.00	6.18
	<i>yahoo.com</i>	60.98	0.89
Social Network/Dating	<i>facebook.com</i>	0.00	0.07
	<i>myspace.com</i>	25.00	65.55
Shopping	<i>ebay.com</i>	0.00	0.10
	<i>amazon.com</i>	0.00	0.78
Corporate	<i>usps.com</i>	0.00	4.66
Overall		33.65	9.09

Table 3: True and false detection rates for web pages in live network data

**Results** The results of our experiment on live data, shown in Table 3, provides some interesting insight into the practicality of identifying web pages in real anonymized traffic. In our results from local testing data collected via automated browsing, we observe that certain categories made up mostly of simple, static web pages (e.g., search engines) provide excellent true detection rates, while web pages whose content changes often (e.g., news web pages) perform significantly worse. Furthermore, categories of sites with complex structure (i.e., many logical servers) generally have exceptionally low false detection rates, while categories of simple sites with fewer logical servers produce extremely high false detection rates. Upon closer examination, not all web pages within a given category perform similarly despite having similar content and function. For instance, *cnn.com* and *nytimes.com* have wildly different performance in our live network test despite the fact that both pages have

rapidly changing news content.

To better understand this difference in our classifier’s performance for different web pages, we examine three features of the page loads in our automated browsing sessions for each site: the number of flows per web browsing session, number of physical servers per web browsing session, and the number of bytes per flow. Our goals lie in understanding (i) why two sites within the same category have wildly different performance, and (ii) why simple web pages introduce so many more false detections over more complex web pages. To quantify the complexity of the web page and the amount of variation exhibited in these features, we compute the mean and standard deviation for each feature across all observations of the web page in our training data, including both simulated cache and unlimited cache scenarios. The mean values for each feature provide an idea of the complexity of the web page. For instance, a small average

number of physical servers would indicate that the web page does not make extensive use of content delivery networks. The standard deviations tell us how consistent the structure of the page is. These statistics offer a concise measure of the complexity of each web page, enabling us to objectively compare sites in order to determine why some are more identifiable than others.

Returning to the difference in true detection rates for *cnn.com* and *nytimes.com*, the front pages of both sites have constantly changing news items, a significant number of advertisements, and extensive content delivery networks. One would expect that since the web pages change so rapidly, that our accuracy in classifying them would be comparable. Instead, we find that we can detect nearly half of the occurrences of *cnn.com* in live network data, while we never successfully detect *nytimes.com*. In Table 4, we see that both web pages have similar means and standard deviations for both flow size and number of physical servers. This similarity is likely due to the nature of the content these two sites provide. However, the number of flows per web browsing session for *nytimes.com* is nearly double that of *cnn.com*. Moreover, *nytimes.com* exhibits high variability in the number of flows it generates, while *cnn.com* seems to use a fairly stable number of flows in all web browsing sessions. This variability makes it difficult to construct high-quality kernel density estimates for the logical servers that support *nytimes.com*, so our detector necessarily performs poorly on it.

Another interesting result from our live network evaluation is that some web page models appear to match well with almost all other pages, and therefore cause an excessive amount of false detections. For instance, Table 3 shows that *yahoo.com* has an exceptionally low false detection rate among all web pages in our live traffic, while *google.com* has one of the highest false detection rates. Both web pages, however, provide adequate true detection rates. In Table 5, we see that *google.com* and *yahoo.com* have very distinct behaviors for each of the features.

The metrics for our three features show that *google.com* transfers very little data, that there is almost always only one physical server in the web browsing session, and that there are normally only one or two flows. On the other hand, *yahoo.com* serves significantly more data, has a substantial number of physical servers, and causes the browser to open several flows per web browsing session. The web browsing sessions for *google.com* and *yahoo.com* both exhibit very little variability, though *yahoo.com* has more variability in its flow sizes due to its dynamic nature. Since both web pages have relatively low variability for all three features, they are both fairly easy for our techniques to detect, which corroborates our earlier claim that *cnn.com* is easy to detect because

of the relative stability of its features. However, since *google.com* is so simplistic, with only a single physical server and very few flows on average, its BBN and KDE models have very few constraints that must be met before the detector flags a match. Hence, many physical servers in a given NetFlow log could easily satisfy these requirements, and this causes the detector to produce an excessive number of false detection. By contrast, the models for *yahoo.com* have enough different logical servers and enough flows per session that it is difficult for any other site to fit the full description that is captured in the BBN and KDE models.

**Discussion** With regard to realistic threats to anonymized network data, these results show that there are certain web pages whose behavior is so unpredictable that they may be very difficult to detect in practice. Also, an attacker has little chance of accurately identifying small, simple web pages with our techniques. Complex web pages containing large content delivery networks, on the other hand, may allow an attacker to identify these pages within anonymized flow traces with low false detection rates. Finally, we have found that an attacker must consider the effects of locality on the training data used to create the target web page models, such as the presence of private caching servers or proxies. These locality effects adversely influence the true detection rates, but they might be overcome through diversification of the training data from several distinct locations. It is unclear how this diversification would affect the performance of our techniques.

When evaluating the threat that our web page identification attack poses to privacy, it is prudent to consider the information an attacker can reliably gain, possible practical countermeasures that might hamper such attacks, and the overarching goals of network data anonymization. With the techniques presented in this paper, an attacker would be able to create profiles for specific web pages of interest, and determine whether or not at least one user has visited that page, as long as those target web pages were of sufficient complexity. While the attacker will not be able to pinpoint which specific user browsed to the page in question with the technique presented in this paper, such information leakage may still be worrisome to some data publishers (e.g., web browsing to several risqué web pages).

There are, however, practical concerns that may affect the attacker's success aside from those described in this paper, such as the use of ad blocking software and web accelerators that dramatically alter the profiles of web pages. These web browsing tools could be used to make the attacker's job of building robust profiles more difficult, as the attacker would not only have to adjust for locality effects, but also for the effects of the particu-

Feature	<i>cnn.com</i>		<i>nytimes.com</i>	
	Mean	Std. Dev.	Mean	Std. Dev.
Number of Flows	18.44	4.21	30.69	10.62
Number of Physical Servers	12.79	2.27	15.32	4.14
Flow Size (KB)	568.20	286.95	692.87	298.73

Table 4: Comparison of mean and std. deviation for features of *cnn.com* and *nytimes.com*

Feature	<i>google.com</i>		<i>yahoo.com</i>	
	Mean	Std. Dev.	Mean	Std. Dev.
Number of Flows	1.73	0.56	9.02	3.02
Number of Physical Servers	1.03	0.17	5.25	1.79
Flow Size (KB)	13.64	10.37	219.51	187.26

Table 5: Comparison of mean and std. deviation for features of *google.com* and *yahoo.com*

lar ad blocking software or web accelerators. Moreover, while our evaluation has provided evidence that certain classes of web pages are identifiable despite the use of anonymization techniques, it is unclear how well the true detection and false detection rates scale with a larger target web page set. Therefore, our techniques appear to be of practical concern insofar as the attacker can approximate the behavior of the browsers and network environment used to download the web page.

## 8 Conclusion

In this paper, we perform an in-depth analysis of the threats that publishing anonymized NetFlow traces poses to the privacy of web browsing behaviors. Moreover, we believe our analysis is the first that addresses a number of challenges to uncovering browsing behavior present in real network traffic. These challenges include the effects of network locality on the adversary’s ability to build profiles of client browsing behavior; difficulties in unambiguously parsing traffic to identify the flows that constitute a web page retrieval; and the effects of browser caching, content distribution networks, dynamic web pages, and HTTP pipelining. In order to accommodate for these issues, we adapt machine learning techniques to our problem in novel ways.

With regard to realistic threats to anonymized NetFlow data, our results show that there are certain web pages whose behavior is so variable that they may be very difficult to detect in practice. Also, our techniques offer an attacker little hope of accurately identifying small, simple web pages with a low false detection rate. However, complex web pages appear to pose a threat to privacy. Finally, our results show that an attacker must consider the effects of locality on the training data used to create the target web page models.

Our results and analysis seem to contradict the widely

held belief that small, static web pages are the easiest target for identification. This contradiction can be explained by the distinct differences between closed world testing and the realities of identifying web pages in the wild, such as browser caching behavior and web browsing session parsing. On the whole, we believe our study shows that a non-trivial amount of information about web browsing behaviors is leaked in anonymized network data. Indeed, our analysis has demonstrated that anonymization offers less privacy to web browsing traffic than once thought, and suggests that a class of web pages can be detected in a flow trace by a determined attacker.

## Acknowledgments

The authors would like to thank Angelos Keromytis, Gabriela Cretu, and Salvatore Stolfo for access to network trace data used in early work on this topic. Also, thanks to our shepherd, Paul Van Oorschot, for providing insightful comments and guidance. This work was supported in part by NSF grant CNS-0546350.

## Notes

<sup>1</sup>Though machine learning techniques are certainly not the only method for handling variability in web pages, their application in this context seems to be intuitive.

<sup>2</sup>Note that even if this assumption did not hold there are still techniques that can be used to infer the presence of HTTP traffic (e.g. based on traffic-mix characteristics).

## References

- [1] BISSIAS, G., LIBERATORE, M., JENSEN, D., AND LEVINE, B. N. Privacy Vulnerabilities in Encrypted HTTP Streams. In *Proceedings of the 5th International Workshop on Privacy Enhancing Technologies* (May 2005), pp. 1–11.

- [2] BREKNE, T., AND ÅRNES, A. Circumventing IP-Address Pseudonymization. In *Proceedings of the 3<sup>rd</sup> IASTED International Conference on Communications and Computer Networks* (October 2005).
- [3] BREKNE, T., ÅRNES, A., AND ØSLEBØ, A. Anonymization of IP Traffic Monitoring Data – Attacks on Two Prefix-preserving Anonymization Schemes and Some Proposed Remedies. In *Proceedings of the Workshop on Privacy Enhancing Technologies* (May 2005), pp. 179–196.
- [4] Cisco IOS NetFlow. <http://www.cisco.com/go/netflow>.
- [5] COLLINS, M. P., AND REITER, M. K. Finding Peer-to-Peer File-Sharing Using Coarse Network Behaviors. In *Proceedings of the 11<sup>th</sup> European Symposium on Research in Computer Security* (September 2006), pp. 1–17.
- [6] COULL, S., WRIGHT, C., MONROSE, F., COLLINS, M., AND REITER, M. Playing Devil's Advocate: Inferring Sensitive Information from Anonymized Network Traces. In *Proceedings of the 14<sup>th</sup> Annual Network and Distributed System Security Symposium* (February 2007). Available at: <http://www.cs.jhu.edu/~fabian/NDSS07.pdf>.
- [7] CRAWDAD: A Community Resource for Archiving Wireless Data at Dartmouth. <http://crawdad.cs.dartmouth.edu>.
- [8] DANEZIS, G. Traffic Analysis of the HTTP Protocol over TLS. Unpublished manuscript available at <http://homes.esat.kuleuven.be/~gdanezis/TLSanon.pdf> as of February 1, 2007.
- [9] FAN, J., XU, J., AMMAR, M., AND MOON, S. Prefix-preserving IP Address Anonymization: Measurement-based Security Evaluation and a New Cryptography-based Scheme. *Computer Networks* 46, 2 (October 2004), 263–272.
- [10] GUPTA, P., AND MCKEOWN, N. Packet Classification Using Hierarchical Intelligent Cuttings. In *Proceedings of Hot Interconnects VII* (1999), pp. 147–160.
- [11] HINTZ, A. Fingerprinting Websites Using Traffic Analysis. In *Proceedings of the 2nd International Workshop on Privacy Enhancing Technologies* (April 2003), pp. 171–178.
- [12] KARAGIANNIS, T., PAPAGIANNAKI, K., AND FALOUTSOS, M. BLINC: Multilevel Traffic Classification in the Dark. In *Proceedings of ACM SIGCOMM* (August 2005), pp. 229–240.
- [13] KIM, M., KANG, H., AND HONG, J. Towards Peer-to-Peer Traffic Analysis Using Flows. In *Self-Managing Distributed Systems, 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management* (October 2003), pp. 55–67.
- [14] KOUKIS, D., ANTONATOS, S., AND ANAGNOSTAKIS, K. On the Privacy Risks of Publishing Anonymized IP Network Traces. In *Proceedings of Communications and Multimedia Security* (October 2006), pp. 22–32.
- [15] KREIBICH, C., AND CROWCROFT, J. Efficient Sequence Alignment of Network Traffic. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement* (October 2006), pp. 307–312.
- [16] LEI, J. Z., AND GHORBANI, A. A. The Reconstruction of the Interleaved Sessions from a Server Log. In *Proceedings of the 17<sup>th</sup> Canadian Conference on AI* (May 2004), pp. 133–145.
- [17] LI, Y., SLAGELL, A., LUO, K., AND YURCIK, W. CANINE: A Combined Conversion and Anonymization Tool for Processing NetFlows for Security. In *Proceedings of Tenth International Conference on Telecommunication Systems* (2005).
- [18] LIBERATORE, M., AND LEVINE, B. Inferring the Source of Encrypted HTTP Connections. In *Proceedings of the ACM conference on Computer and Communications Security* (October 2006), pp. 255–263.
- [19] MARTIN, D., AND SCHULMAN, A. Deanonymizing users of the safeweb anonymizing service. In *Proceedings of the 11th USENIX Security Symposium* (August 2002), pp. 123–137.
- [20] NEAPOLITAN, R. *Probabilistic Reasoning in Expert Systems: Theory and Algorithms*. John Wiley & Sons, Inc., 1989.
- [21] NICKLESS, W., NAVARRO, J., AND WINKLER, L. Combining CISCO NetFlow Exports with Relational Database Technology for Usage Statistics, Intrusion Detection, and Network Forensics. In *Proceedings of the 14th Large Systems Administration Conference (LISA)* (December 2000), pp. 285–290.
- [22] ØVERLIER, L., BREKNE, T., AND ÅRNES, A. Non-Expanding Transaction Specific Pseudonymization for IP Traffic Monitoring. In *Proceedings of the 4<sup>th</sup> International Conference on Cryptology and Network Security* (December 2005), pp. 261–273.
- [23] PANG, R., ALLMAN, M., PAXSON, V., AND LEE, J. The Devil and Packet Trace Anonymization. *ACM Computer Communication Review* 36, 1 (January 2006), 29–38.
- [24] PANG, R., AND PAXSON, V. A High-Level Environment for Packet Trace Anonymization and Transformation. In *Proceedings of the ACM Special Interest Group in Communications (SIGCOM) Conference* (August 2003), pp. 339–351.
- [25] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. *SIGCOMM Computer Communication Reviews* 33, 1 (January 2003), 5964.
- [26] PREDICT: Protected Repository for the Defense of Infrastructure Against Cyber Threats. <http://www.predict.org>.
- [27] SCOTT, D. *Multivariate Density Estimation: Theory, Practice, and Visualization*. John Wiley & Sons, New York, 1992.
- [28] SILVERMAN, B. *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC, 1986.
- [29] SINGH, S., BABOESCU, F., VARGHESE, G., AND WANG, J. Packet Classification Using Multidimensional Cutting. In *Proceedings of ACM SIGCOMM* (August 2003), pp. 213–224.
- [30] SLAGELL, A., LI, Y., AND LUO, K. Sharing Network Logs for Computer Forensics: A New Tool for the Anonymization of NetFlow Records. In *Proceedings of Computer Network Forensics Research Workshop* (September 2005).
- [31] SPILIOPOULOU, M., MOBASHER, B., BERENDT, B., AND NAKAGAWA, M. A Framework for the Evaluation of Session Reconstruction Heuristics in Web-Usage Analysis. *INFORMS Journal on Computing* 15, 2 (April 2003), 171–190.
- [32] SUN, Q., SIMON, D. R., WANG, Y., RUSSELL, W., PADMANABHAN, V. N., AND QIU, L. Statistical Identification of Encrypted Web Browsing Traffic. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2002), pp. 19–31.
- [33] XIE, Y., SEKAR, V., MALTZ, D., REITER, M. K., AND ZHANG, H. Worm Origin Identification Using Random Moonwalks. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy* (May 2005), pp. 242–256.

# THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

## ***Membership Benefits***

- Free subscription to *login:*, the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications: see <http://www.usenix.org/membership/specialdisc.html>

For more information about membership, conferences, or publications, see <http://www.usenix.org>.

## **SAGE, a USENIX Special Interest Group**

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at <http://www.sage.org>.

## **Thanks to USENIX & SAGE Corporate Supporters**

Ajava Systems, Inc.	Hewlett-Packard	rTIN Aps
Cambridge Computer Services, Inc.	IBM	Sendmail, Inc.
cPacket Networks	Infosys	Splunk
DigiCert® SSL Certification	Intel	Sun Microsystems, Inc.
EAGLE Software, Inc.	Interhack	Taos
FOTO SEARCH Stock Footage and Stock Photography	MSB Associates	Tellme Networks
Google	NetApp	UUNET Technologies, Inc.
GroundWork Open Source Solutions	Oracle	VMware
	Raytheon	Zenoss
	Ripe NCC	

